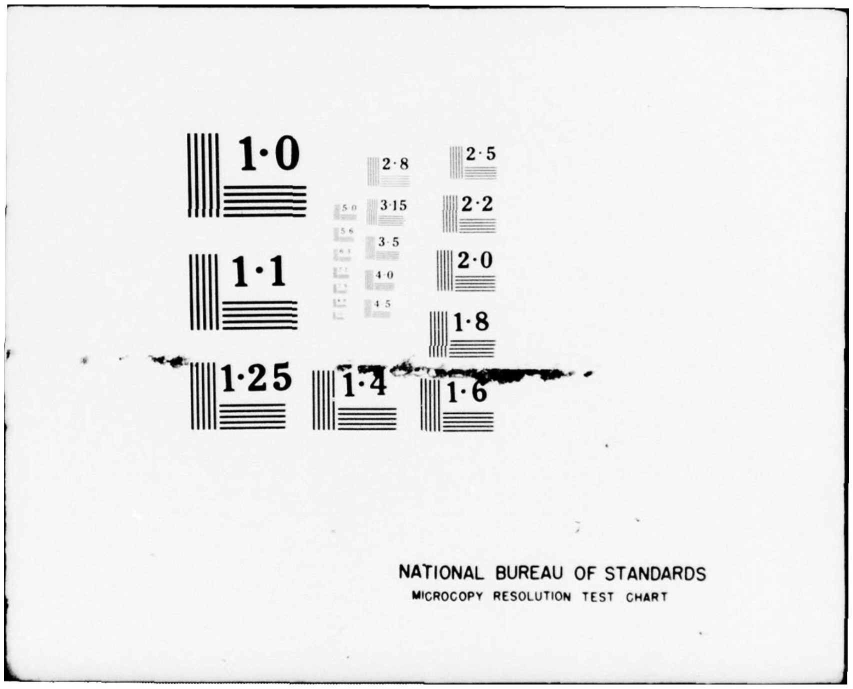
AD-A044 213   ROME AIR DEVELOPMENT CENTER GRIFFISS AFB N Y   F/G 9/2
A COMPARISON OF SEQUENTIAL AND ASSOCIATIVE COMPUTING OF PRIORIT--ETC(U)
AUG 77 B M LANDSON

UNCLASSIFIED   RADC-TR-77-221   NL

1 OF 3
ADA
044213


1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
4·5
1·1
1·8
1·25
1·4
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

12

RADC-TR-77-221
In-house Report
August 1977

A COMPARISON OF SEQUENTIAL AND ASSOCIATIVE COMPUTING OF PRIORITY QUEUES WITH APPLICATIONS TO DISCRETE SIMULATION

Major Barry M. Landson

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been submitted by the author to Syracuse University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Systems and Information Science.

The motivation of the research was to explore the possibilities of simplifying and improving the non-numeric aspects of discrete simulation by associative processing. To that end, the author gratefully acknowledges the guidance of Dr. Robert Sargent.

APPROVED: [illegible]

BRUCE W. PURDY
Lt Col, USAF
Chief, Image Systems Branch

APPROVED: Howard Davis

HOWARD DAVIS
Technical Director
Intelligence and Reconnaissance Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief
Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 14 RADC-TR-77-221 ✓ — 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER: 9
4. TITLE *(and Subtitle)*: 6 A COMPARISON OF SEQUENTIAL AND ASSOCIATIVE COMPUTING OF PRIORITY QUEUES WITH APPLICATIONS TO DISCRETE SIMULATION.	5. TYPE OF REPORT & PERIOD COVERED: Final rept., In-house
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): 10 Major Barry M. Landson	8. CONTRACT OR GRANT NUMBER(s): N/A
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Rome Air Development Center (IRR) Griffiss AFB NY 13441	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F/62440001 17 ΦΦ 16
11. CONTROLLING OFFICE NAME AND ADDRESS: Same	12. REPORT DATE: 11 August 1977
	13. NUMBER OF PAGES: 198
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same 12 201p.	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Associative Computing
Simulation and Modeling
Queueing

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Over the past fifteen years, a number of authors have studied methods of improving data management in discrete simulation. These methods have dealt with the random access memory. Over the same period, a number of other authors have studied various forms of associative architecture. The following research combines both fields of study to focus on discrete simulation data management. The specific concern is priority queues and time flow mechanisms.

(over)

309 050 LB

The approach used is to specify three forms of associative memories and their associated algorithms and then compare them with a random access memory which uses contemporary algorithms. The associative and random access memories were formed as hardware models upon which were overlaid algorithmic models of priority queue procedures. The measurement used was total time for comparable tasks. The models were developed in parametric form so that the parametric interrelationships and design tradeoffs could be studied.

The following is a summary of the results. The direct algorithmic map-over of random access memory algorithms to associative architecture was unsatisfactory. The algorithmic procedures had to be reconstituted with parallel processing in mind. Further, the traditional time flow mechanisms could not be used and a new time flow mechanism, called fixed increment minimum value, was developed. With the advent of new algorithmic procedures, the various forms of the associative memories did show distinct processing potential based on parametric values established for the computer models. The advantage or disadvantage of the associative architectures with respect to the random access was most sensitive to the ratio of the memory speeds and the processing width or degree of parallelism in the associative cases. The research indicates that hybrid associative memories are superior to straight associative memories for discrete simulation.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF SYMBOLS

A	Dummy variable for Algorithms S_1 and S_2.
a	Random access memory sort-in factor.
AM	Associative memory.
AMA	Associative memory address field.
$\overrightarrow{AMA}$	Width of AMA.
AML	Associative memory - Lewin.
b	Random access memory priority sort-in factor.
c	Random access memory list depth for first success.
CDLLL	Circular double linked linear list.
C_j	Counter one for queues.
C'_j	Counter two for FIFO queues ($C'_j = C_j$ for LIFO queues).
CTR	Counter field.
d	Random access memory - number of successful nodes.
DLLL	Double linked linear list.
e	Random access memory - number of successful nodes.
EBW	Equivalent bit width.
$\overrightarrow{f}_{i,k}$	Width in bits of field k in composite node cycle i.
FIFO	First in first out queue discipline.
FIMV	Fixed increment minimum value time flow mechanism.
$\overrightarrow{f}_{S\ i,k}$	Width in bits of search field k in composite node cycle i.
FTI	Fixed time increment time flow mechanism.
$\overrightarrow{f}_{T\ i,k}$	Width in bits of transfer field k in composite node cycle i.

LIST OF SYMBOLS (continued)

l_i	List i.
$\overrightarrow{l_i}$	Length of list i in nodes.
LIFO	Last in first out queue discipline.
LLINK	Left link field for CDLLL.
LN	List name field.
$\overrightarrow{LN}$	Width of LN field in bits.
M	Number of consecutive nodes to be retrieved.
M_a	Length of associative memory in words for AM/RAM or AM/RAM/AML architecture.
M_{al}	Length of Lewin's memory in long words.
M_l	Length of memories in words for AM or RAM architecture.
M_r	Length of RAM for AM/RAM or AM/RAM/AML architecture.
M_w	Width of memories in bits (except AML).
PK	Primary key.
$\overrightarrow{PK}$	Width of primary key in bits.
RAM	Random access memory.
RLINK	Right link field in CDLLL.
RNA	RAM node address field.
$\overrightarrow{RNA}$	Width of RNA in bits.
RNAL	RAM node address - Lewin.
$\overrightarrow{RNAL}$	Width of RNAL in bits.
SCC	Simulation control code.
SISD	Single instruction single datum architecture.

LIST OF SYMBOLS (continued)

SIMD	Single instruction multiple data architecture.
SK	Secondary key.
$\overrightarrow{SK}$	Width of secondary key in bits.
SLLL	Single linked linear list.
ST_i	Simulation time i.
T_A	Random access memory instruction time.
T'_A	Associative memory and associative memory - Levin auxiliary instruction time.
TFM	Time flow mechanism.
TK	Tertiary key.
$\overrightarrow{TK}$	Width of tertiary key in bits.
TLLL	Triple linked linear list.
T_M	Random access memory instruction time.
T'_M	Associative memory and associative memory - Levin instruction time.
TOE	Time of entry field.
TT_i	Total time for i^{th} composite node cycle.
T_1, T_2 T_3, T_4	Reference times for fixed increment minimum value time flow mechanism.
VTI	Variable time increment time flow mechanism.
$\lceil y/x$	Next highest integer value of y/x.
α	Ratio of T_M/T'_M.
β	$= \beta_1 = \beta_2$ for $T_A = T'_A$.
β_1	Ratio of T_A/T'_M.

LIST OF SYMBOLS (continued)

β_2	Ratio of T'_A/T'_M.
τ	Dummy variable for Algorithm S_2.
γ	Associative processing width factor.
Δt	Time increment for time flow mechanisms.
δ	Associative transfer factor.
η_i	Dummy variable for Algorithm A_1.
$\rightarrow$	Bit width or list length.

CHAPTER I

INTRODUCTION

The purpose of discrete event simulation is to study systems which change at discrete points in time. Doing discrete simulation with a computer requires methods of creating, storing, retrieving and consuming (destroying) information within the computer during the course of time. The consequence of this dynamic information processing is that a major problem arises in the generally excessive length of computer time needed to conduct discrete simulations [22]. The solution to this problem involves utilizing efficient methods for non-numerical dynamic information processing, of which the major task is the efficient manipulation of priority queues, not only because priority queues are found in the type of system studied in discrete simulation, but, more importantly, because priority queues are the method used to move discrete simulations through time so that they can represent time varying systems.

Specifically the research considers alternative forms of associative memories which may be used to implement priority queues found in discrete simulation. A comparison is then made with a random access memory to determine under what conditions a particular memory is more efficient for a particular priority queue. The consideration of associative memories requires that the concept of data parallelism be considered--that is, the number of items and the amount of information within each one that can be acted upon simultaneously by a single computer instruction.

The overall approach used in the research is to postulate in parametric form four computer architectural models--three associative and one random access. Overlaid on these models are parametric representations of the priority queues used in discrete simulation. The total computer time required for each architecture to perform each priority queue computational task is then determined. This is followed by a comparison of the performance of the various architectural types in terms of comparing total time for similar tasks.

The research draws on four major areas of previous research. The first area is discrete simulation. The best single reference for the general state of the art in discrete simulation is Fishman [22]. He covers both the numeric and non-numeric processes of discrete simulation and gives special attention to the analysis of discrete simulations.

A subset of discrete simulation is time flow mechanisms, the manner in which discrete simulations move through time. In this area there are several authors who have studied more efficient implementations. Their research is discussed in Chapter II.

The second area involves general random access memory algorithmic procedures. The authority in this area is Knuth, in his four-volume _The Art of Computer Programming_ [36, 37, 38, 39]. His work is heavily referenced in Chapter II and Chapter III.

The third area is associative architecture. The two authors in this area upon whom this research relies quite heavily are Feng [17, 18, 19, 20, 21] for associative algorithmic procedures and Flynn [23] for methodology, measurements and perspective. The work of these authors and others is discussed in Chapters II and III.

The fourth area deals with the intersection of the previous three and is the focus of this research. There are four pieces of research in this area, which are discussed chronologically. The first was the doctoral thesis of Kroft in 1969 [41]. He designed and demonstrated an associative memory for list processing. List processing is very close to the priority queue processes used in discrete simulation, and much of the early work in discrete simulation was a direct result of list processing techniques. The second was a paper by Posdamer and others in 1971 [60] which advanced several ideas regarding the use of associative architecture for discrete simulation. The remaining two were doctoral theses and both were published in 1972. The first, by DeFiore [12], was concerned with the use of the associative memory for data management purposes. His work involved the use of non-numeric processes, primarily in the area of fixed data relationships as opposed to dynamic. The last work, by Davis [11], was concerned with discrete simulation from the point of view of reducing the amount of time spent in branching within various algorithms. His overall scheme did include an associative memory, but the main thrust is toward fully parallel processing. In the specific area of priority queues and associative memories no previous detailed research work is known.

The dissertation is composed of five chapters. Following the introduction, background information in discrete simulation is provided in Chapter II. The computer research models and tasks are discussed in Chapter III. The results are covered in Chapter IV; and conclusions and recommendations are in Chapter V. Two appendices are also included which cover introductory material on associative processing and detailed non-numeric algorithms used in the research.

CHAPTER II

DISCRETE SIMULATION

2.1. Introduction

This chapter is intended to introduce the subject and describe the functions that are involved in discrete simulation. Tutorial material along with examples can be found in the references from which the material in this chapter was taken [16, 22, 26, 53].

Discrete simulation as used in this research deals with the study of a dynamic system whose behavior can be represented as a time sequence of discrete changes which occur according to some stochastic process. Discrete simulation does not deal with the system directly but with an abstraction of the system called a model. The model and the means of moving it through time and measuring its behavior (the simulation mechanism) are represented in the computer. The complexity of the computer representation is a function of the complexity of the problem and the system under study.

Included in this chapter is a discussion of discrete simulation methodology, the functions involved in discrete simulation, the manner in which the functions become computer representations, and, in detail, one of the prime functions that permit a discrete simulation to move through time in a computer environment. As such the general background to computational discrete simulation is provided to serve as a preamble to the specific research.

2.2. Discrete Simulation Methodology

Discrete simulation methodology is shown schematically in Figure 1. A discussion of methodology is included here not only to provide additional information on discrete simulation but also to put the intended research into better perspective by showing specifically where this research applies.

The first step in the methodology is to formulate the problem. It is at this step that a decision is made whether or not to use discrete simulation as the problem solving method. Careful investigation is required before a commitment because simulation should only be used if an analytical technique can not be applied economically to the problem. Formulation of the problem is usually augmented by the data collection step. The data are initially used for the decision and, if the choice is discrete simulation, they are used for model formulation (including parameter estimation) and evaluation. This last step is a paper and pencil operation where the researcher attempts to formulate what he considers to be an accurate abstraction (model) of a real or proposed system. Accurate is used here in the sense that the model must properly imitate the system for the purpose desired and hence must be sufficiently complete. If the model is too complete, going into too much detail, there will be excessive cost incurred in model implementation, validation, and execution. The completeness of the model then becomes a matter of judgment,

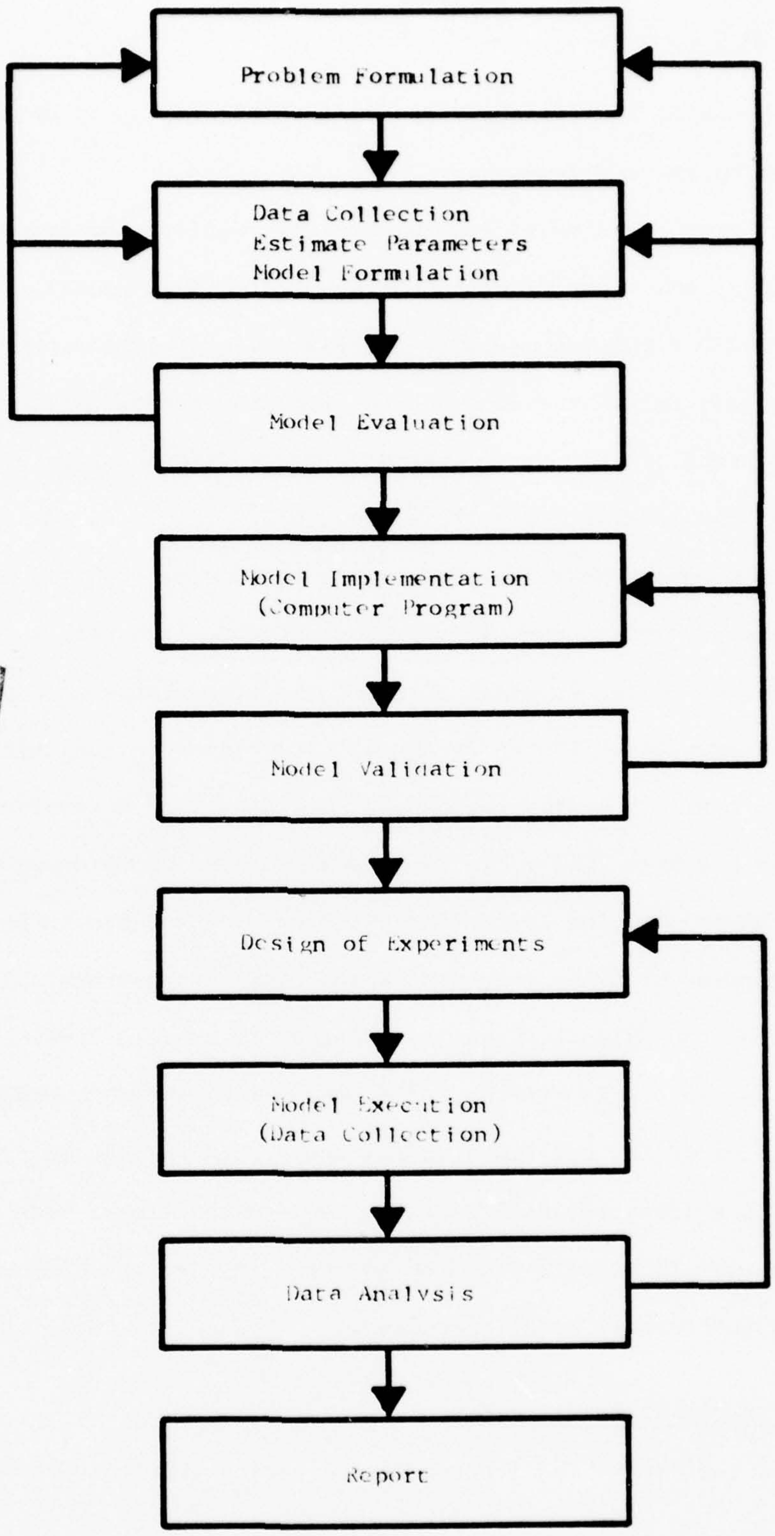


Figure 1. Discrete Simulation Methodology

which is one of the reasons that discrete simulation is more of an art than an exact science.

The next step, model implementation, requires the researcher to transfer his paper model to the computer. This usually involves implementing a complex computer program where special attention must be paid to sequencing and describing the model. This task is somewhat facilitated by the availability of discrete simulation packages and languages [5, 35, 61, 62, 67]. This step and the subsequent one of model validation are particularly difficult, since the researcher must insure that the model behavior is correct with respect to the original system. The correctness of the model can sometimes be validated by comparing selected outputs with the original system; however, a frequent use of discrete simulation is to study a system that is in the planning stage or to study some aspect of an existing system for which there is little information.

Assuming that the preceding steps have been successfully completed, the balance of the methodology is concerned with the formal design of experiments, data generation, and data analysis, with the report as the final step. The feedback lines in Figure 1 reflect the iterative nature of the total methodology. The research itself affects those steps which involve the computer and, in particular, model execution.

2.3. Functional Requirements

The functional requirements for discrete simulation are shown in Figure 2. These requirements are divided into control and support functions. The control functions are synonymous with the simulation mechanism. Many of these ideas are discussed by Pritsker [62].

Control Functions

F_0 - Discrete Simulation Sequencer. The role of the discrete simulation sequencer is to move the simulation through the various control functions or states according to the requirements of the user. This function may not exist as a separate program and most commonly occurs as a linkage among the other control functions.

F_1 - Initialization. The initialization function sets up the initial conditions for both the control functions and the model. This is referred to in the literature as setting up the model or simulation runs and amounts to setting up all the data values used to start the simulation.

F_2 - Time Flow Mechanism (TFM). The time flow mechanism has three nominal purposes: maintaining the simulated time or clock, selecting the next potential happening within the model, and maintaining a list of future potential happenings [52, 71]. The TFM does not necessarily cause a state change within a model, but creates the possibility of a state change according to preprogrammed instructions. The method used to implement this function is critical to the computational efficiency of the

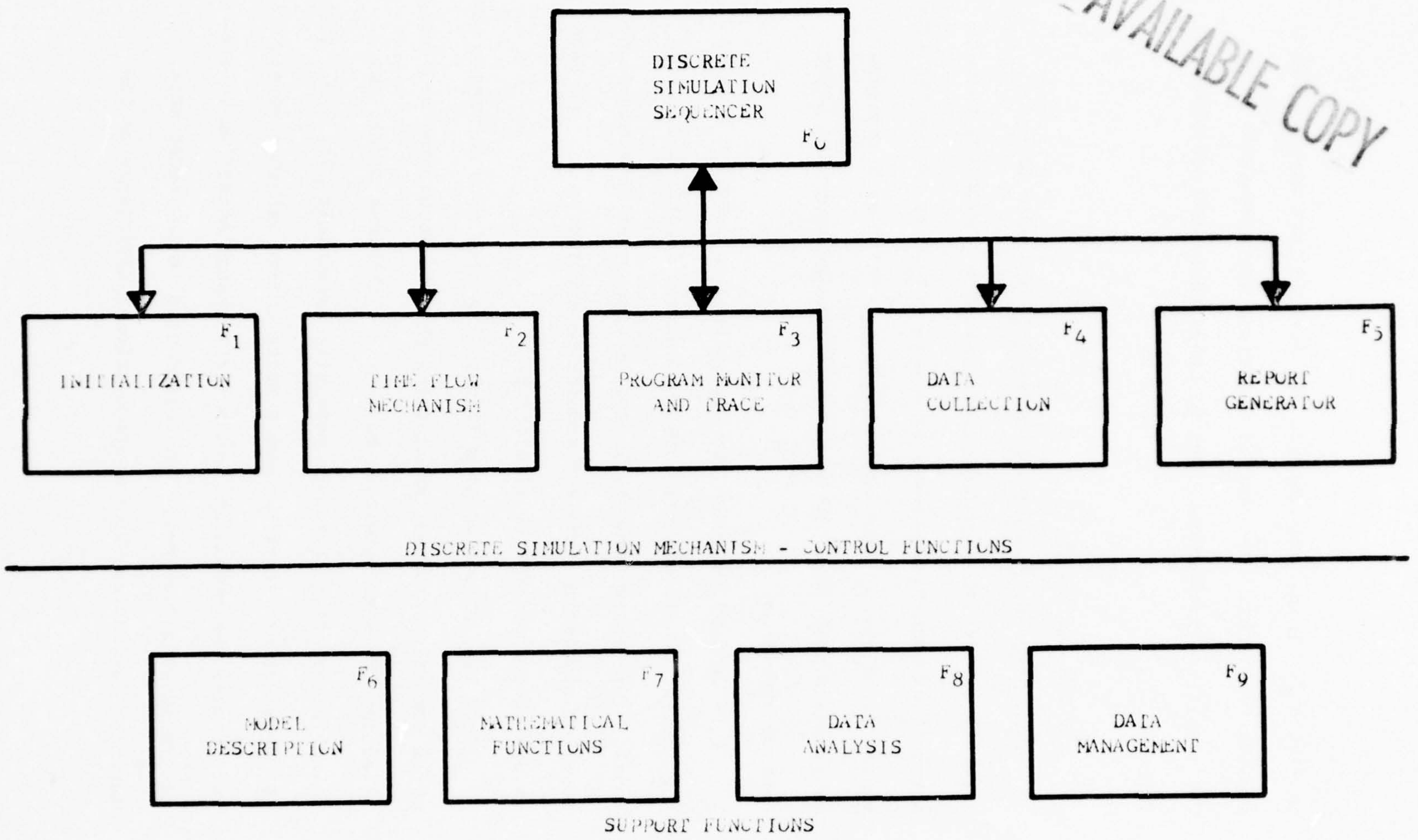

DISCRETE SIMULATION SEQUENCER F_0
F_1 INITIALIZATION
F_2 TIME FLOW MECHANISM
F_3 PROGRAM MONITOR AND TRACE
F_4 DATA COLLECTION
F_5 REPORT GENERATOR
DISCRETE SIMULATION MECHANISM - CONTROL FUNCTIONS
F_6 MODEL DESCRIPTION
F_7 MATHEMATICAL FUNCTIONS
F_8 DATA ANALYSIS
F_9 DATA MANAGEMENT
SUPPORT FUNCTIONS

Figure 2. Discrete Simulation Functions for Model Execution

simulation and is problem dependent [8, 44, 50, 71]. The normal process for the TFM is to select a potential next state change and then activate that part of the model description program responsible for determining whether or not that particular state change can take place at this time. If it can, the state change is implemented by the model description function. At the completion of the state change, control is returned to the TFM by the model description function along with a list of future potential state changes. The TFM places any new potential state changes in its main list, possibly sorting them on time, updates the clock if appropriate, and then chooses the next potential state change which starts the TFM cycle again. Therefore a discrete simulation exists by virtue of the cooperation between the TFM and the model description function.

F_3 - Program Monitoring and Trace. This function is activated only at the preprogrammed request of the researcher, either to report exceptional conditions during the course of the simulation or to provide snapshots of a partial time history of the model. Its primary use is as a run time diagnostic tool.

F_4 - Data Collection. The data collection function keeps track of the number of times and amount of time that the model is in any one of a number of preselected states. This function makes available the statistical data necessary for the design of experiments analysis.

F_5 - Report Generator. The report generator supplies a history of the discrete simulation including any statistical output requested. Its primary function is to support the data generation phase of the simulation.

Support Functions

F_6 - Model Description. This function contains all the information necessary to imitate the actual system accurately. The representation of this function within the computer is quite difficult and there appears to be no single best way to do it.

F_7 - Mathematical Programs. The prime role of this function is to provide the random numbers and variates necessary to reproduce the stochastic nature of the system. Other programs may involve trigonometric formulae and minimums and maximums.

F_8 - Data Analysis. This function includes all the activities necessary to reduce the statistical data produced by the discrete simulation.

These statistical activities would normally include time series analysis [3], design of experiments [15, 33, 34], and other special activities. These activities are not normally a part of a simulation package or language but are handled separately after data collection has taken place.

F_9 - Data Management. This function is concerned with the efficient management of data stored in the computer. This function may exist explicitly, as in data management systems or list processing languages, or implicitly within other functions. It is the former that will be used in this research where each other function is considered to have one or more complex data activities. Later this function will be referred to as node management.

It should be clear that with the possible exception of the TFM and model description functions there is nothing specific to discrete simulation that might not be used in other computer applications.

2.4. Information Structures

There are three major factors that must be considered in moving from a paper and pencil representation to a computer implementation of the problem. These are the data (amount and type), the data structure (data organization or relationships), and the operations (functions) germane to the data structures. When these three items are represented within a particular computer they are referred to as the data, the storage structure and the algorithms, discussed below. Several authors have considered the general and specific problems associated with the efficient combination of the three items [1, 6, 13, 14, 69]; however, in more recent years many authors [22, 34] have turned to Knuth [36, 38, 39] as the reference authority on such matters. In particular, Fishman [22] points out that the operations and data structures considered as the building blocks for discrete simulation are presented in Knuth. It is for this reason and others discussed later that Knuth will serve as the principal reference for this work.

Before Knuth can be used as the direct reference it is necessary to make some definitions and based on these definitions relate Knuth's work to other references. Chapin [6] and D'Imperio [13] make a definite distinction between a data structure and a storage structure. They point out that a data structure is a way of looking at or arranging the relationships among the data so that the resulting structure 'makes sense' with respect to the problem at hand. In terms of discrete simulation methodology, this corresponds to the model formulation step where the model description function is initially set up. The formulation of a data structure then is a precomputer activity. Once a data structure is established it must be mapped onto the existing storage system within the computer. This storage system may differ from computer to computer and may consist of core memory, magnetic tape or others. After the mapping, the data structure is then referred to as a storage structure. Depending on the storage system, there may be several alternative storage structures for a particular data structure. The selection of a particular storage structure is based on the operations to be carried out as part of the problem solution. These operations correspond in the most general sense to the various functions involved in discrete simulation discussed in the previous section. In particular, the operations supporting the data management function in the management of various storage structures are of interest in this research. Implementing a particular function may require one or more algorithms. In the case of data management in discrete simulation, several algorithms are involved. Since an algorithm is simply a formal procedure it is possible for it to exist outside a computer; however, in this research it will be assumed that it is a computer procedure.

Knuth integrates the dual concept of data structure and operation into the single concept of an information structure. An information structure then has form (data structure) and purpose (function). The function can be represented by operations upon the data structure. Ultimately an information structure exists within the computer as a storage structure and one or more algorithms. An efficient implementation on a computer of an information structure then requires matching the storage structure with the algorithms, which in general is both problem and computer dependent.

Storage Structures

Storage structures can be classified into linear and nonlinear structures, according to Knuth [36]. The linear structures can be further partitioned into linear lists and linear arrays, where the latter is an extension of the former. Fishman [22], Gordon [26], and Emshoff and Sisson [16] illustrate that the linear list is currently the major storage structure used in discrete simulation. As such, it will be the main concern of this research.

The following is Knuth's definition of a linear list:

> A linear list is a set of $n \geq 0$ nodes X(1), X(2), ..., X(n) whose structural properties essentially involve only the linear (one dimensional) relative positions of the nodes: the facts that, if $n > 0$, X(1) is the first node; when $1 < k < n$, the kth node X(k) is preceded by X(k-1) and followed by X(k+1); and X(n) is the last node. [36]

A node is the basic component of a storage structure, and consists of one or more words of computer memory. The words are contiguous and each is partitioned into named parts called fields.

There are three common storage structures used to represent linear lists, at least within conventional computer memories. These are sequential, single linked, and double linked. Examples are shown in Figure 3. The first method stores information sequentially within the computer. The next two methods permit nodes to be stored randomly within the memory, but the list, instead of being maintained by contiguity, is maintained by links which are implanted within each node. In the singly linked case the link to the second node in the list is stored in the first node and so forth. The links in this case are simply the memory addresses of subsequent nodes. In the doubly linked case each node maintains the link or address of both its predecessor and successor in this list. In both the linked cases the lists can be made circular, so that the last node in the list has the address of the first node. This permits continuous access throughout the list. The circular double linked linear list is of particular interest here because it is the one most often used to implement the information structures encountered in discrete simulation [16, 22, 26, 36].

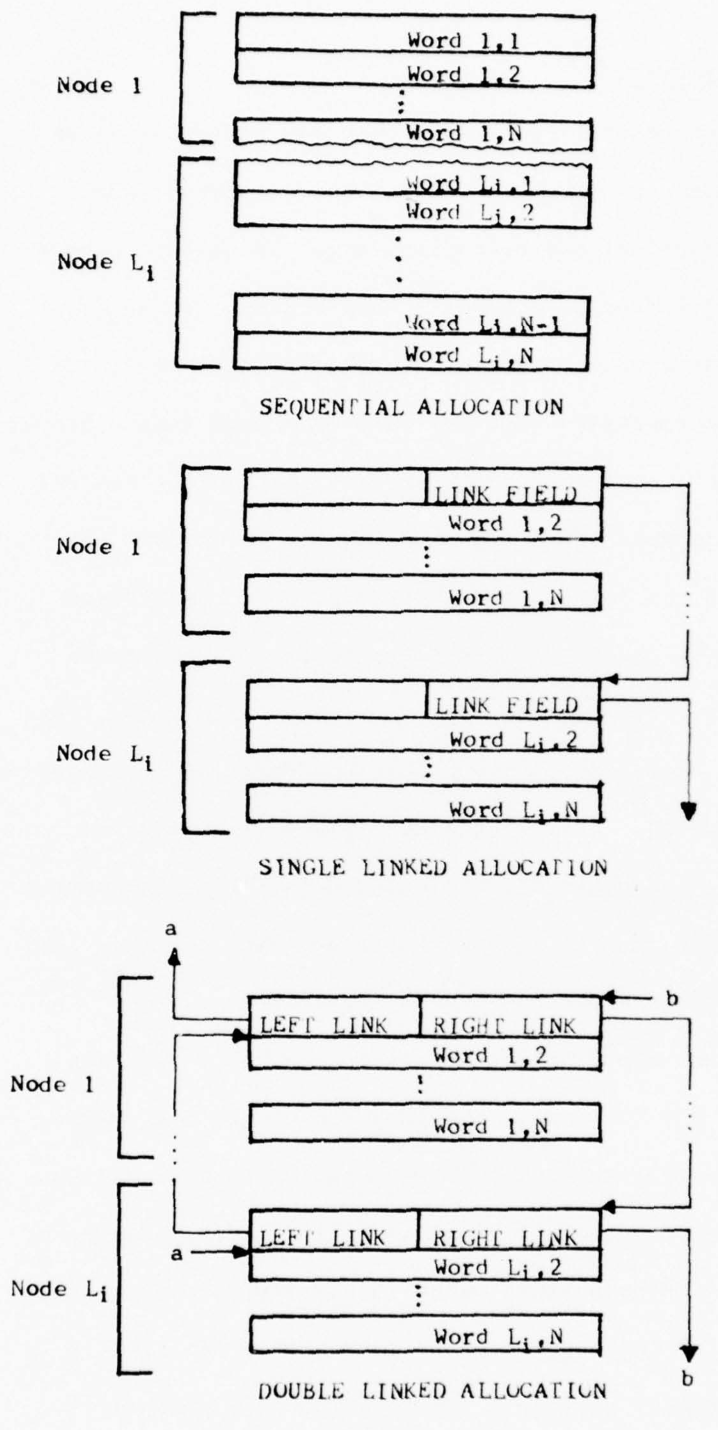


Figure 3. Linked Allocation

Operations/Algorithms

There are nine general operations that can be performed on linear lists. These are shown in Table 1 and are taken from Knuth [36]. Although all the operations might be used in discrete simulation, some are used more heavily than others. These are discussed below and are noted by an asterisk in the table.

The insertion operation involves putting a node into a linear list. If the list uses a sequential storage structure, then all the nodes below the node to be inserted must be moved down one node position. If the list is represented by a linked storage structure, insertion is the process of establishing the proper linkage with the predecessor and successor nodes within the list.

The deletion operation is the companion operation to insertion. Here a node is removed. If the node was from a list using a sequential storage structure all the nodes below it must be moved up one node. If the list is linked the predecessor and successor nodes must be relinked after the removal operation.

In both the above operations, it is necessary to know the exact location of the K^{th} node. The location can be determined by prefacing the insertion with a sort operation which ranks the nodes according to the values in some specified field. Based on this ordering the K^{th} node field value is usually less than or greater than that of the node to be inserted. For deletion it might only be necessary to remove the first or last node in a ranked list so that K is determined. If the deletion is to be of

1. Gain access to the Kth node of the list to examine or change the contents of its field.

*2. Insert a new node just before the Kth node.

*3. Delete the Kth node.

4. Combine two or more lists (linear) into a single list.

5. Split a list into two or more lists.

6. Make a copy of a linear list.

7. Determine the number of nodes in a linear list.

*8. Sort the nodes of a list into ascending order

*9. Search the list for the occurrence of a node with a particular value in some field.

*Indicates operations receiving heaviest use in discrete simulation.

Table 1. Linear Information Structure Operations

(from Knuth [36])

a node which is not first or last then a search operation must take place. This amounts to going through the list one node at a time until the proper node is found, based on some search criteria.

The four operations just discussed--insert, delete, sort and search--all are involved with changing the content of the list. In discrete simulation the contents of many lists are changing with time as the simulation progresses. This activity is what enables discrete simulation to represent dynamic systems.

As mentioned earlier, for efficient computer implementation it is necessary to match the storage structure and the algorithms. In discrete simulation, this has come to mean that the principal working storage structure is the circular double linked linear list. This is so for the general and specific reasons listed below, paraphrased from Knuth [36]. (Note particularly the relation of item two to the dynamic nature of the lists in discrete simulation mentioned earlier.)

1. Linked allocation does require more memory space than sequential; however, in the circular double linked linear list (CDLLL) case of discrete simulation this usually amounts to one word per node, split into two address fields, where the node may contain typically up to ten words. In other words one would incur a ten per cent storage disadvantage in order to obtain the benefits listed below.

2. Deletion within the CDLLL is particularly fast because the deleted node contains both successor and predecessor addresses within itself. Thus deletion requires only a few index operations. In the sequential case several nodes may have to be moved to fill the gap caused by deletion.

3. Insertion is also generally simpler when linkage is used.

There are two additional operations which are necessary to support a highly dynamic environment. These are the processes of allocation and deallocation, which are involved with the management of physical storage such as computer words. Allocation forms physical storage into nodes and then makes the nodes available for use. When a node is no longer needed, such as following a deletion operation, the deallocation process makes the physical storage used by that node available for future allocation.

For each operation mentioned there is an algorithm which implements it. In this case there is a one to one correspondence between operation and algorithm. There are in general several choices as to which algorithm should be used to implement a particular operation. Chapter III discusses the selection of the algorithms considered in this research.

2.5. Priority Queues and Time Flow Mechanisms

A priority queue is an information structure that occurs in all discrete simulations. It is the information structure most commonly considered to be the key to improving the computational efficiency of discrete simulation because of its relationship to TFM's. For this reason it is the information structure that is the subject of this research. Knuth points out that priority queues exist frequently in other computational applications outside discrete simulation [36]. This research applies to these other situations if the specific priority queues are similar.

To support the preceding remarks, the following discussion will delineate priority queues and the various roles they play in discrete simulation.

A priority queue is a combination of a waiting line and a service discipline. It has form (data structure) based on the relationship of the items (data) waiting for service. It has function in that the purpose of a priority queue is to provide service to the items waiting according to the service discipline (operation).

There must be some selection process by which the next item is selected for service. This is called the priority or queue discipline. The priority discipline must provide the rules for making the following two decisions [32]:

1. Which unit to select for service once the server is free to take up the next unit.

2. Whether to continue or discontinue the service of the unit being serviced.

There are two special cases of priority queues, hereafter called queues. These are the first in first out (FIFO) and the last in first out (LIFO) disciplines. These queues will be studied separately because of their special properties when represented by storage structures and algorithms.

Priority queues occur in discrete simulation in two major ways. The first is that frequently the system under study involves priority queues. Where such a case exists the model description function, which eventually must be represented by storage structures and algorithms, must imitate the priority queue. The second occurrence of priority queues is in the time flow mechanism function. Since the main purpose of the TFM is to select the next potential happening (event) within the model from a waiting line of choices it is by definition a priority queue. This selection process is done by a primary key that contains the simulation time for the next potential happening. The priority queue discipline is then lowest key value (earliest time).

Several researchers have pointed out that the TFM is of major concern in efficient computer implementation of discrete simulation and have also gone on to study various ways of improving the TFM. Among these researchers are Conway [8], Lave [44], Nance [52], Wickham [71], Vaucher and Duval [69], and Morgan and Siegel [50]. Since this research will address time flow mechanisms and in particular Morgan and Siegel's work, additional background is provided.

Time flow mechanisms (at least since Conway) have been classified in two different types, known as the variable time increment (VTI) and the fixed time increment (FTI), with the exception of Wickham's work. Morgan and Siegel suggested an adaptive TFM which switched between the two different types of TFM's in accordance with a certain decision rule. In all cases except Morgan and Siegel, the intention has been a priori to select or match an efficient storage structure with the attendant algorithms for the particular discrete simulation.

Variable Time Increment

The variable time increment (VTI) time flow mechanism is a procedure whereby the TFM advances time to the next potential state change. As Conway [8] and Morgan and Siegel [50] point out, this method is less time-consuming if the state changes occur relatively infrequently with respect to the unit of time used in the simulation. The very nature of this procedure also requires that the future state changes be schedulable--that is, if some entity is due to change state at some future time, that the time is known or can be calculated without regard for a state change involving some other entity at a time prior to the actual state change. As an example, the arrival of a patient at the emergency room, considered as an activity external to the model (exogenous activity), can be calculated without regard for other happenings within the model. Such an arrival would constitute a new item in the system.

Fixed Time Increment

The fixed time increment (FTI) time flow mechanism relies on advancing the simulation clock by a unit of time and then conducting a comparison with each of the entities to see if any next potential happenings fall within the last time increment. In terms of storage structures and algorithms, the FTI approach is equivalent to a search of a random (non-ranked) list for all nodes that are less than or equal to some time value.

Lave points out that there is a time error and possibly a precedence error due to the fact that one does not know exactly when the state change took place, only the interval within which it occurred [44]. Gafarian and Ancker [25] conducted a study in comparing the relative efficiency of the VTI and FTI methods with regard to estimating the expected output of the system and found that because of the time error information is always lost in FTI time flow mechanism simulators. The main advantage to the FTI method is that it is frequently faster than VTI under conditions of dense event changes.

Dynamic Time Flow Mechanism

Morgan and Siegel's TFM switched between the VTI and FTI methods described above. This was done based on a look ahead procedure which measured the upcoming density of future state changes. Based on a decision rule the TFM used the FTI method for dense state changes and the VTI for sparse state changes. The purpose of this approach was to increase the efficiency of the TFM by minimizing the time for the TFM algorithm. A TFM method is discussed later that incorporates Morgan and Siegel's philosophy but with a better efficiency.

CHAPTER III

COMPUTER RESEARCH MODELS AND TASKS

3.1. Introduction

The overall research methodology is shown in Figure 4. This figure will serve as the basis of discussions for Chapters III and IV. The figure also serves as a means of tying together the concepts of this research as discussed below.

The starting point for the research methodology is the selection of information structures to be studied. The rationale for the selection of priority queues and their relationship to discrete simulation was discussed in Chapter II. In particular there are ten cases of priority queues considered in the research. Four of the cases fall under the definition of queues and the remaining six are priority queues. In the latter category, the first three cases under priority queues deal specifically with priority queues used for time flow mechanisms. The remaining cases deal with general priority queues that are used in discrete simulation. All these cases are listed at the bottom of Figure 4.

An information structure is one or more operations (algorithms) operating on a data structure (storage structure). The intersection of a particular storage structure and a set of algorithms constitutes an information structure. However, the intersection of a different storage structure and a different set of algorithms (detailed instruction procedure) can define the same information

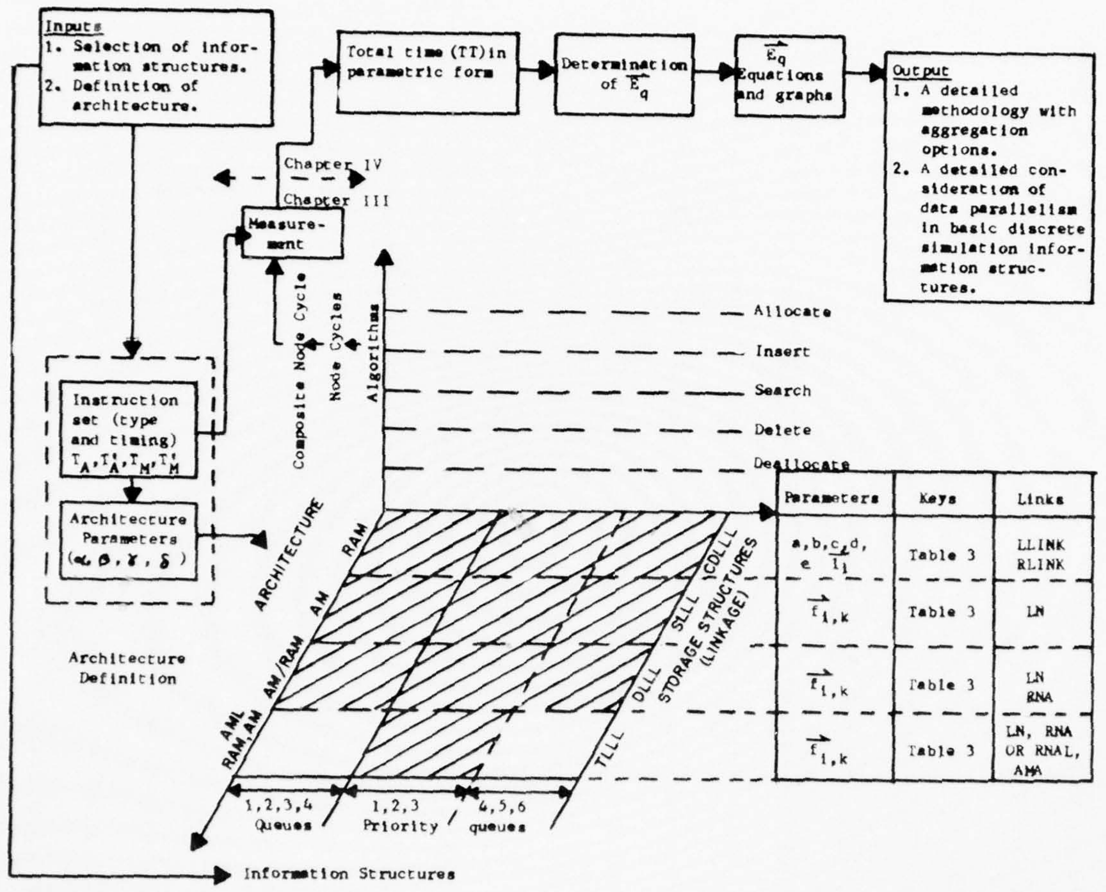

Inputs
1. Selection of information structures.
2. Definition of architecture.
Total time (TT) in parametric form
Determination of $\vec{E_q}$
$\vec{E_q}$ Equations and graphs
Output
1. A detailed methodology with aggregation options.
2. A detailed consideration of data parallelism in basic discrete simulation information structures.
Chapter IV
Chapter III
Measurement
Composite Node Cycle
Node Cycles
Algorithms
Allocate
Insert
Search
Delete
Deallocate
Instruction set (type and timing) T_A, T'_A, T_M, T'_M
Architecture Parameters ($\alpha, \beta, \gamma, \delta$)
Architecture Definition
ARCHITECTURE
RAM
AM
AM/RAM
AML
RAM, AM
CDLLL
SLLL
DLLL
TLLL
STORAGE STRUCTURES (LINKAGE)
Parameters
Keys
Links
a, b, c, d, e $\vec{l_i}$
Table 3
LLINK RLINK
$\vec{f}_{i,k}$
Table 3
LN
$\vec{f}_{i,k}$
Table 3
LN RNA
$\vec{f}_{i,k}$
Table 3
LN, RNA OR RNAL, AMA
1,2,3,4 Queues
1,2,3 Priority
4,5,6 queues
Information Structures

Figure 4. Detailed Research Methodology

structure. In effect there are different ways to implement an information structure, particularly if the architecture (instruction set) is also permitted to vary. To place a meaningful bound on the scope of the research, a single storage structure was chosen for each architecture that seemed to be an efficient match for that architecture, based on some experiments.

The second input to the research methodology is the selection of the architecture to be used for comparison. In addition to the random access memory which was used as a reference, three other architectures involving associate memories are used. The architectures are shown along the slanting left-hand axis of Figure 4. They are the random access memory (RAM), the associative memory (AM), the associative memory used in conjunction with a random access memory (AM/RAM), and finally an associative memory coupled with a random access memory with a special associative search memory used as an auxiliary memory to the first two (AM/RAM/AML). These memories are defined through a parameterized instruction set (type of instruction and instruction timing) which in turn leads to a parametric representation of the architecture. These two steps are shown in the dotted architecture definition block in Figure 4. In addition the instruction set definition was also used to help define the measures used for architectural comparison.

As mentioned above, a single storage structure was used for each architecture. Therefore opposite the architecture axis and parallel to it are shown the storage structures. These are respectively the circular double linked linear list (CDLLL), the single linked linear list (SLLL), the double linked linear list (DLLL), and the triple linked linear list (TLLL).

Shown to the right of the storage structure axis is a small table keyed to the specific architecture which lists keys, links, and parameters. In the link column are the specific names of the links which form the storage structure. The keys complement the links in that where the links identify a specific list and its current organization of the stored data, the keys permit the specific identification of a node and hence permit the selection of specific nodes or the ordering of nodes. The ordering is then the current link structure. The links and keys are covered more fully when Table 3 is discussed later. The parameters, also discussed later, permit the behavior of the various storage structures to be studied as part of the overall study of the information structure. The parameters represent the current storage state of the queue or priority queue.

To complete the discussion of the information structures the five algorithms germane to the ten information structure cases are shown on the vertical axis. This is a slightly different set of algorithms (operations) from what is discussed in Chapter II. This is because the sort activity has been folded into the insert algorithm, since sort does not apply for all architectures where insert does. The intersection then of the five algorithms with each storage structure (and hence architecture) forms an information structure and in particular one specific information structure (queue or priority queue) for each of the ten cases. Altogether, the research explores fifty-seven situations. The information structures and the architectures to be compared based on them have been mated parametrically. This completes the first part of the methodology.

The second part of the research methodology is concerned with the acquisition of comparison data on the architectures. The first step is the definition of node cycles. A node cycle can be thought of as a completed action involving a node in an information structure. This concept is based on the intimacy between a node and the data stored therein. In discrete simulation the data and the node usually have the same lifetime; therefore it is possible to talk about a node cycle instead of a data cycle. The two completed actions used for comparison in the research are the creation and storage of data (node) within a storage structure and the removal and destruction of the data (node) at some future time. These two actions maintain the dynamic information structures of this research. Because of variations arising from measuring these two actions separately, a composite node cycle was defined which concatenates the two actions into what is later described as a birth and death process for nodes. This approach permits studying typical life cycles of nodes based on the storage structure parameters mentioned earlier.

The next step is measurement. The measure used in this research is the total computation time for the composite node cycle of the nodes within each situation studied. The second part of the research methodology permits the collection of quantitative data on the behavior of the information structures.

The third portion of the research methodology is concerned with the formal comparison of the architectures. The problem here is that without certain assumptions the associative and random access architectures do not easily permit direct comparison. The principal alleviating assumption to this problem is that the associative memory has full word operations just as does the typical random access memory. In short this means that a single word compare in a random access memory would take no longer than a multiple word compare in an associative memory. Current commercial implementations of associate memories do not use full word compares partly because this results in increased manufacturing problems and hence costs, and partly because not all algorithms lend themselves easily to such an implementation. One case in point is the minimum search algorithm. Since the minimum search forms such an integral part of maintaining priority queues in discrete simulation, the alleviating assumption mentioned above could not be used directly.

The alternate procedure used for comparison was to let the processing width (the number of bits simultaneously active in a word) in the associate memory vary, where algorithms permitted, from a single bit up to a full word. The result of this procedure is that a trade off can now be made parametrically between the processing width for associative memories and the equivalent total time necessary for the random access memory. This idea is formalized in the equivalent breakeven bit width $(\overrightarrow{Eq})$ equations of Chapter IV. The results of Chapter IV then indicate under which conditions in terms of processing width the associate architectures excel for a particular composite node cycle representing a particular information structure. The overall result of this methodology is twofold. There is first a detailed approach to doing comparisons where the degree of aggregation can be controlled, since it is possible to study the detailed algorithms intimately or to throw away information selectively. There is also the ability to study the suitability of using various forms of associative architecture for the implementation of representative priority queues found in discrete dynamic simulation.

This completes the formal research methodology. The remainder of Chapter III is partitioned into three sections. In the first section the computer research models, including architecture and storage structures and the static portion of the information structures and architectures, are covered. In the next section research algorithms or the active portion of the information structures are discussed. This includes algorithms, search keys and parameters as they apply to the information structures. Finally there is a section on measurements. Figure 4 should be referred to often as reading progresses.

3.2. Computer Research Models

The purpose of studying the behavior of information structures with models is to have a controlled method for delineation of the architecture and a method of comparison. In particular the models specified here can be programmed with a particular workload and therefore offer the opportunity of a detailed analysis that may not be available when working with a more abstract model. This approach permits the consideration not only of the specific task, but also of the overhead necessary to prepare the data for that task. This last point is significant in that Flynn [23] points out that this overhead can significantly affect the relative performance of parallel architecture computers.

Architecture

The first step in setting up the models is the definition of the architecture. This is equivalent to the definition of the instruction set (Table 2). The instruction set is used because it is the basis of implementation of the algorithms and therefore serves as the basis for measurement since timings can be conveniently assigned to groups of instructions.

To establish the instruction set on a sound basis for this and future comparisons, a well established sequential computer model was selected as the basis for all the memory systems. This model is the pedagogical computer MIX, created by Knuth for his _Art of Computer Programming_ series [36, 37, 38, 39]. This

Instructions	Remarks	Instruction Timing			
		MIX-RAM	MIX-AM	MIX-AM/RAM	MIX-AM/RAM/AML
<u>Load</u>					
LDA ADR,I(F)	Add L suffix	T_M	T_M'	T_M'	T_M'
LDX ADR,I(F)	for AML	T_M	T_M'	T_M'	T_M'
LDi ADR,I(F)		T_M	T_M'	T_M'	T_M'
<u>Store</u>					
STA ADR,I(F)	Add L suffix	T_M	T_M'	T_M'	T_M'
STX ADR,I(F)	for AML	T_M	T_M'	T_M'	T_M'
STi ADR,I(F)		T_M	T_M'	T_M'	T_M'
STZ ADR,I(F)	Store zero	T_M	T_M'	T_M'	T_M'
<u>Address Transfer</u>					
INC_ M	M is the	T_A	T_A'	T_A'	T_A'
DEC_ M	value to be	T_A	T_A'	T_A'	T_A'
ENT_ M	added/entered	T_A	T_A'	T_A'	T_A'
<u>Jump</u>					
JA_		T_A	T_A'	T_A'	T_A'
JX_		T_A	T_A'	T_A'	T_A'
JI_		T_A	T_A'	T_A'	T_A'
J _		T_A	T_A'	T_A'	T_A'
JMI_(A,B, or C)	Match Ind	N/A	T_A'	T_A'	T_A'
JLI_(A,B, or C)	Jumps	N/A	T_A'	T_A'	T_A'
<u>Compare</u>					
CMPA I(F)	**	T_M	$\lceil w/\gamma \rceil T_M'$*	$\lceil w/\gamma \rceil T_M'$*	$\lceil w/\gamma \rceil T_M'$*
CMPX I(F)	**	T_M	"	"	"
CMPi I(F)	**	T_M	"	"	"
CMP/MIN I(F)		N/A	$w\ T_M'$	$w\ T_M'$	$w\ T_M'$
CMPL/MIN I(F)	Lewin's alg	N/A	N/A	N/A	$(2M-1)T_M'$
<u>Miscellaneous</u>					
MOVE		$(1+M)T_M$	$(1+M)T_M'$	$(1+M)T_M'$	$(1+M)T_M'$*
MOVE/	PIO	N/A	$\lceil w/\delta \rceil T_M'$	$\lceil w/\delta \rceil T_M'$	$\lceil w/\delta \rceil T_M'$

* $w = \bar{f}_{i,k}$ = width of ith field in kth list.
  γ = processing width in bits.
  δ = transfer width in bits.
** For the detailed algorithms in Appendix B,
  / indicates associative compare with prior reset (e.g. CMPA/EQ I(F)).
  // indicates associative compare without prior reset (e.g. CMPA//EQ I(F)).

Table 2. Instruction Sets

selection immediately yielded the following benefits:

1. MIX is fully documented in terms of an instruction set and instruction timing.

2. The RAM algorithms for manipulating the information structures of interest in the research are documented on the MIX computer.

3. The MIX computer (in terms of its instruction set) permitted a reasonably straightforward conversion from a sequential computer to an associative computer with the addition of or replacement by an associative memory.

To explain this last step and to introduce the associative architecture, some background is necessary. Two distinct memories are considered in this research, the random access and the associative. The random access memory is characterized by the fact that it may access any location in memory randomly one location at a time. An associative memory may access all selected locations at once although not necessarily the full word at each location. In this sense the associative memory is able to associate (compare) some input with all candidate stored words (as in a search) and annotate all matching words. The associative memory may not consider all parts of all words in a single memory activation or interrogation because of the associated cost of the hardware [30, 69]. The associative memory can address in its simplest form one bit in each selected word, usually the same bit. This method of implementation is sometimes known as the bit slice or bit mode [21].

Flynn [23] formalizes the preceding discussion by pointing out that the sequential computer is a single instruction single datum (SISD) form of architecture while the associative memory falls into the class of single instruction multiple data (SIMD) architecture. Therefore the only difference in theory is that the associative memory exhibits a form of data parallelism in that it can operate on several pieces of data simultaneously. In practice, conversion of MIX to an associative memory did not require a change of instructions, only a redefinition with respect to multiple data. (A general example of associative memories is given in Appendix A.)

The four memory organizations of Figure 5 may be considered as follows. Figure 5A represents the standard MIX computer documented in Knuth's books with the random access memory. Figure 5B represents the MIX computer with an associative memory substituted for the random access memory. Notice that with the exception of the busy bit, the response store register, and the match indicator, both memories contain the same number of words, M_1, each with the same bit width, M_w. That means that each memory configuration has the same number of data words of the same width available for storage.

Note that this is different from STARAN [63] which has very long word lengths, i. e., 256 bits. The intent here is to make the memories as comparable as possible and thus to isolate the single datum from the multiple data for measurement purposes. In performing the actual research it is assumed that each word holds only one field so that the actual value of M_w is not important.

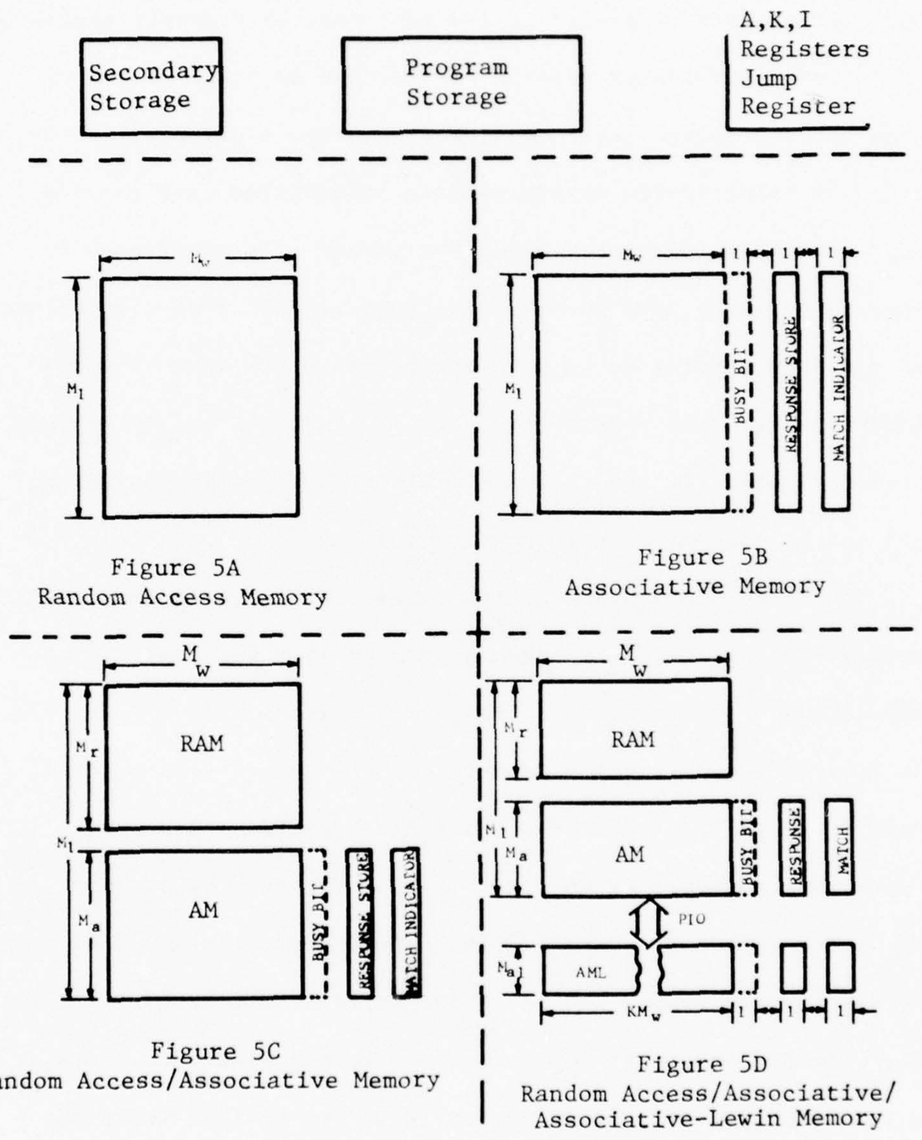

Secondary Storage
Program Storage
A,K,I Registers Jump Register
M_w
M_l
1
BUSY BIT
RESPONSE STORE
MATCH INDICATOR
Figure 5A Random Access Memory
Figure 5B Associative Memory
M_r
M_a
RAM
AM
RESPONSE
MATCH
PIO
AML
M_{al}
KM_w
Figure 5C Random Access/Associative Memory
Figure 5D Random Access/Associative/ Associative-Lewin Memory

Figure 5. Research Architecture

The exception, noted later, is Lewin's memory.

Figure 5C represents a combination of the first two memory configurations such that the total storage remains the same. This configuration was suggested by the fact that in discrete simulation a node usually contains several fields, but only one or two of these contain search keys used for retrieving a particular node, while the other fields merely contain information that is used after a node is selected. Since the amount of storage used for other than search keys can be significant in discrete simulation, and since in general it is more expensive to use data in associative rather than random form [30, 70], why not split the node between associative and random storage with appropriate linkage? This is the essence of this configuration.

The last memory organization, shown in Figure 5D, was included because of the minimum search problem mentioned in the introduction. It is based on Lewin's algorithm [19, 45, 72, 73]. The minimum search problem is particularly critical in cases 1, 2 and 3 for priority queues. The basic organization of this fourth configuration is the same as that for the third except that a parallel input/output (PIO) channel has been added between the associative memory and the associative memory-Lewin, labelled AML. The AML memory is M_{al} words long by KM_w bits wide where K is some integer. M_{al} would typically be ten or less words and K would be between five and ten for most discrete simulation applications. The essence of the algorithm is that it guarantees that M distinct records can be identified (selected) in order, ascending or descending, in not more than 2M-1 memory interrogations. The benefit of this auxiliary memory is that of providing rapid ordered retrieval, but it requires a more complex memory, which is more costly. This is offset to some extent in that it probably need not be very large.

The second part of the memory definition is the specification of the instruction set. The instruction set for all four memory configurations is given in Table 2. The load, store, address transfer and the first four jump instructions operate in the same way for all four memories. They are documented and discussed in Knuth's books [36, 37, 38, 39] and will not be further elaborated here. Load and store instructions pertaining exclusively to the AML memory are suffixed by L, otherwise all memory regardless of block is considered contiguous in addressing. This set of conditions implies that these associative memories may be addressed either in a data parallel mode or by conventional addressing. For a further discussion of these ideas consult Stone [66], Shooman [64, 65], Wolinsky [74], Brotherton [4] and Rudolf [63]. Wolinsky [74], Parhami [57] and Hyde [31] provide good backgrounds for associative memories and related technology.

The two remaining jump commands JMI_ and JLI_ are used to test the match indicators associated respectively with associative memory and the AML memory.

The major difference among the memories occurs with the compare instruction. The normal compare instruction for MIX permits a value stored in any register to be compared with any specified word in memory. The associative counterpart to this is that any value in any register may be compared in a data parallel mode throughout the memory. A distinction is made however by assigning a processing width factor (γ) to the associative memory so that the total amount of time necessary for an associative memory to complete a compare is a function of both the number of bits used to represent a search or a compare field (w) and the simultaneous processing width (γ). This is indicated by the $\lceil w/\gamma$ symbol in the timing columns for comparand compares where $\lceil\;$ stands for next highest integer value of the quotient if there is a remainder. Gamma then represents the number of bit slices active simultaneously for a given memory interrogation. The companion to gamma is delta (δ) which represents the number of bit slices transferred simultaneously from the associative memory to the special Lewin's memory. In both cases the maximum value of either gamma or delta is M_w.

The general compare discussed above is also referred to as a comparand compare. In effect a reference value is set up in a register and all selected words are compared with it according to the compare criterion, greater than, less than, et cetera. Each word is compared independently of the others so that gamma can be meaningfully introduced. There is another type of compare which is a noncomparand compare. This is the search for minimum or maximum. Such a search can not be conducted independently on each word. Therefore simply increasing the processing width is not appropriate. Feng [19] discusses various types of minimum and maximum searches and points out that to achieve the fastest ordered retrieval (selecting members from a list in order) Lewin's algorithm is appropriate. In this research gamma will be one in the minimum search algorithm of Feng [21], or Lewin's algorithm will be used. This therefore bounds the two documented extremes for "parallel" minimum searches in terms of the research. The minimum compare is very important since it is the main associative instruction for implementation of priority queues.

The timings shown in Table 2 are of two types, T_A and T'_A, which represent the instruction time for any non-memory instruction in the random access and associative cases, respectively, and T_M and T'_M, which represent the compare time for one word in the random access case or γ bit slices in the associative case.

No attempt was made to introduce exotic or special purpose hardware with the exception of Lewin's algorithm, and that was introduced as an auxiliary memory. The various memories were constrained to be alike as much as possible including the instruction set.

Storage Structures

In general a node contains three types of information: information which relates one node to another, called linkage information or simply links; selection information (keys) which provides the means to select or identify a node (usually uniquely) from other nodes; and other information appended to the node. The last type of information is not considered here because it is not germane to the research other than to serve as a reason for investigating the AM/RAM architecture. Selection information is discussed under algorithms.

Storage structures are usually classified in random access memories by the type of linkage information contained in a node [6, 7, 13, 14, 36]. As explained earlier for linear lists, the common types of structures are sequential, single, and double linked. The idea of classifying storage structures by linkage information was extended to associative architectures in Figure 4, where each architecture has associated with it a particular linkage structure. Shown to the right in the figure of the linkages are the actual named links for each architecture. As was pointed out previously an information structure may be represented by several alternative storage structures. However for this research a single storage structure was selected for each architecture that best seemed to fit the architecture. The various storage structures will now be discussed. This will complete the model description made up of the architecture and the storage structure.

The primary storage structure for the random access memory in discrete simulation is the circular double linked linear list (CDLLL). This particular structure was discussed in Chapter II. The implementation of this structure is usually by two links, known as the left link (LLINK) and the right link (RLINK). Although the

CDLLL is usually used for all lists in discrete simulation it is possible and practical to use a singly linked linear list (SLLL) to maintain the list of available storage. This is a list made up of the empty nodes not currently in use in the discrete simulation. Knuth uses this procedure and therefore in this particular case the SLLL is used in the research to maintain the list of empty nodes. This choice is also favorable to the RAM.

In the pure associative memory it is also adequate to use only a SLLL. The particular single link used in the associative memory storage structure is the list name (LN) link. Therefore each list needed for a discrete simulation stored in an associative memory has one link which uniquely identifies all nodes belonging to that list. To conduct a particular operation on a particular list it is necessary only to preface that operation with an equal search based on LN which in turn will annotate all current nodes belonging to that list.

The associative memory used in conjunction with a random access memory uses two links and is therefore classified as being double linked. The first link, LN, is used in the same way as in the associative only case, and a second link is added to permit splitting the node into two parts. The second link is the RAM node address link (RNA) and as its name implies it is the address of a node in random access memory that contains appended information to the node stored in the associative memory. The introduction of the second link permits the number of words that are needed in the associative memory to be reduced to just the number needed to locate a node uniquely. It should be pointed out that the RNA field could just as easily be the address of a secondary storage location such as a disc. Since the RNA points to a node in RAM it resides physically in the associative memory.

The storage structure for the fourth architecture is triply linked. As in the previous case, LN and RNA are used in the same way. The third link is the associative memory address (AMA). The AMA link is a self link that is transferred to the AML memory when the node is transferred and points back to the associative memory node location. This permits a node selected in the AML to be first referenced back to the associative memory via the AMA link and then back to the appended information in the RAM via the RNA link. This saves time and storage by avoiding the transfer of appended information into the AML. This triply linked structure assumes that Lewin's memory is used as an auxiliary memory to the AM/RAM. In certain cases Lewin's memory is used directly without being activated by a prior transfer from the AM portion of the AM/RAM memory. In such cases, where the new node is placed directly within the Lewin's memory, the appended information is placed within the RAM and a RAM node address - Lewin (RNAL) link is substituted for the RNA link. In such a case the AMA link is not needed, reducing that particular node to two links. It is anticipated that such a condition would represent a small portion of the total node usage in discrete simulation.

In all cases the storage structures are linear lists in the sense that the links only refer to information within the same list. There are no links stored between information in different lists. An additional assumption is also made about the storage structures. That is, that a uniform node size is used throughout a particular simulation. This means that all nodes in all lists use the same number of words. As Knuth implies, this greatly simplifies the implementation of allocation and deallocation algorithms for the RAM. This assumption does not seem to represent a practical limitation in discrete simulation with regard to wasted storage since it is frequently the case that the number of words needed per node in various lists is within a few words of each other. Additionally the comparison of nonuniform node sizes within the context of this research is a significant undertaking and it is suggested in Chapter V that this be considered as a follow-on research topic.

3.3. Research Algorithms

The purpose of this section is to describe those computational activities which will overlay the models. The computational activities or research algorithms will then form the model driving function. The response of the model to this workload will be measured in the manner described in the next section. The algorithms are shown in Figure 4. These algorithms, for reasons described later, are then merged into node cycles and then further to a composite node cycle. It is this last entity, the composite node cycle, that serves as a formal model task to be measured.

Algorithms in General

This section is not intended to discuss algorithms in detail, since that is done in Appendix B. Rather the purpose here is to describe and classify the types of algorithms used in discrete simulation. As discussed earlier, discrete simulation as considered in this research is a dynamic activity. As such it requires the creation, storage, retrieval and eventual use of data. After data is used it is generally destroyed. Since the data is stored in nodes in such a fashion that all the data within the node are usually used at one time, the behavior of data within discrete simulation may be likened to a stochastic birth and death process, where the nodes are spawned by some stochastic mechanism (born),[1] exist for some period of time (live) and then die. To carry this idea somewhat further, the three portions of the life cycle will be used in an analogous fashion to outline the necessary algorithms.

The birth process involves the creation of the data, the allocation of the node which is to receive the data from a list of available nodes, the placement of the data into the node, and finally the insertion of the node into some sort of storage structure. The birth process, then, can be defined by four types of algorithms: creation, allocation, placement and insertion. Of these, creation is usually a numeric process and is therefore not the subject of this research. Of the remaining three, placement

[1]One can not always guarantee formally that the node process is in fact a continuous parameter Markov chain with homogeneous transition intensities [58].

has been divided between allocation and insertion since the different memory organizations require different contents within the nodes to support the particular information structure. Therefore to account for these differences a family of allocation and insertion[2] algorithms have been set up, each appropriate to the information structure and memory organization, so that the effect of the additional data required within the node can be measured. The additional data is defined as that data which is not common to all memory organizations for a particular information structure. For instance, where the random access memory by itself uses links and the associative memory does not, there are different algorithms for allocation and insertion. At the outset of the research a separate algorithm for each memory organization and storage structure was provided. However, because of some commonality, this was reduced to a subset of the original algorithm list.

The life process is merely the storage over time of a node within a storage structure. Although there are no algorithms directly concerned with this phase, the fact that nodes are present within a storage structure has a bearing on the measurable behavior of the algorithms associated with birth and death.

The eventual utilization and then destruction of the information are grouped under the death of the node because they are adjacent in time. The death of the node begins when it is selected by some retrieval process. Retrieval deletes the node from the storage structure. This is followed by the removal of the information from

[2]Insertion includes sorting or ranking the storage structure for priority queues for the RAM architecture.

the node and then the deallocation of the node or return to a pool of available nodes. The death process can be defined then by four algorithms: deletion (retrieval), removal of information, usage of information and deallocation. Of these, usage is usually a numeric process and is not the subject of this research. Removal, like placement, is divided between deletion and deallocation, again for the same reasons. This means that death can be defined by the two algorithms of deletion and deallocation. The exception to this occurs when considering cases four, five and six under priority queues. Here there is the additional step of search. This is so because deletion in the RAM architecture normally assumes that there is a real or implied order to the storage structure from which the node is deleted. This is not true in cases four, five and six, so the deletion is prefaced by a search algorithm.

Search Keys - General

In terms of specific implementation, certain keys are necessary. These are listed in Table 3 and discussed below. The keys are discussed first along with a description of the various priority queues considered in the research. This discussion will center around Table 3.

Search Keys - Queues

Cases one and two in Table 3 are for the FIFO discipline, while cases three and four are for LIFO. As indicated in the table, there are no keys for the RAM architecture, and therefore cases one, two, three and four are the same. This is because in the RAM

Architecture

Information Structure		RAM		AM		AM/RAM		AM/RAM/AML		*
		Links	Keys	Links	Keys	Links	Keys	Links	Keys	Remarks
Queue	1	LLINK/RLINK	N/A	LN	PK(TOE)	LN/RNA	PK(TOE)			FIFO
	2	LLINK/RLINK	N/A	LN	PK(CTR)	LN/RNA	PK(CTR)			FIFO
	3	LLINK/RLINK	N/A	LN	PK(TOE)	LN/RNA	PK(TOE)			LIFO
	4	LLINK/RLINK	N/A	LN	PK(CTR)	LN/RNA	PK(CTR)			LIFO
Priority Queue	1a	LLINK/RLINK	PK	LN	PK	LN/RNA	PK	LN/RNA(RNAL)/AMA	PK	SN
	1b	LLINK/RLINK	PK	LN	PK	LN/RNA	PK	LN/RNA(RNAL)/AMA	PK	MIN
	1c	LLINK/RLINK	PK	LN	PK	LN/RNA	PK	LN/RNA(RNAL)/AMA	PK	MNN
	2a	LLINK/RLINK	PK/SK	LN	PK/SK	LN/RNA	PK/SK	LN/RNA(RNAL)/AMA	PK/SK	SN
	2b	LLINK/RLINK	PK/SK	LN	PK/SK	LN/RNA	PK/SK	LN/RNA(RNAL)/AMA	PK/SK	MIN
	2c	LLINK/RLINK	PK/SK	LN	PK/SK	LN/RNA	PK/SK	LN/RNA(RNAL)/AMA	PK/SK	MNN
	3a	LLINK/RLINK	PK/SK	LN	PK/SK/TK	LN/RNA	PK/SK/TK	LN/RNA(RNAL)/AMA	PK/SK/TK	SN
	3b	LLINK/RLINK	PK/SK	LN	PK/SK/TK	LN/RNA	PK/SK/TK	LN/RNA(RNAL)/AMA	PK/SK/TK	MIN
	3c	LLINK/RLINK	PK/SK	LN	PK/SK/TK	LN/RNA	PK/SK/TK	LN/RNA(RNAL)/AMA	PK/SK/TK	MNN
	4	LLINK/RLINK	PK	LN	PK	LN/RNA	PK			
	5	LLINK/RLINK	PK	LN	PK	LN/RNA	PK			
	6	LLINK/RLINK	PK	LN	PK	LN/RNA	PK			

*SN = single node
MIN = multiple identical nodes
MNN = multiple non-identical nodes

Table 3. Information Structures, Links and Keys

storage structure queues are maintained by physical order, hence searches and thus search keys are unnecessary. The insertion algorithm simply places the next node first in the list and the deletion algorithm removes the appropriate node in the list--the one with the longest (FIFO) or shortest (LIFO) waiting time in the list. In the three associative architectures the lists are maintained in a random order and therefore searches are necessary. This means that a primary key upon which to search must be provided. Two different primary keys are studied under the FIFO discipline and two under the LIFO discipline. The first key is a time of entry (TOE) key (cases one and three) which reflects the simulation clock at the time the entry is made. The second is a counter (CTR) key (cases two and four). An index register is incremented and decremented by one for each insertion or deletion. The value of the index register(s) is then compared against the CTR key. It is possible to use a list of contiguous numbers for queues with FIFO or LIFO disciplines because insertion and deletion occur in a regular manner. In the FIFO case two index registers are used, whereas for LIFO only one is needed.

<u>Search Keys - Priority Queues</u>

The prime interest in studying priority queues for discrete simulation is that they are the information structure used to implement time flow mechanisms. Cases one, two and three under priority queues along with their individual subcases are introduced specifically for this purpose. In the random access architecture it is assumed that the priority queue is based on a rank ordering of the nodes on the primary key. Priority queues also come up in the context of general searches. In this role priority queues are used to find the first node that meets some specified key value, to find all the nodes that meet a specified key value, or to find the minimum or maximum node based on a specified key. In this situation it is assumed that either the list is unranked or ranked on some key other than the one currently participating in the search. These latter three cases are covered in priority queue cases four, five and six.

Search Keys - Priority Queues, Cases 1, 2, 3

The implementation of the TFM priority queue is somewhat more complex than implementing other types of priority queues, first because there is frequently more than one key to deal with, and second because of the manner in which new nodes arrive in the list. In the latter area, new nodes represent new future potential state changes within the simulation model. As such their primary key is a future simulation time. Since simulations do not back up, there is a guarantee that the value of the primary key is always greater than or equal to the present simulation time. It is also usual that the value of the primary key for new nodes entering the TFM list places them near the bottom of the ranked list, and there are usually several nodes at the top of the list which always have key values less than the new node key value. This creates a situation where the top part of the list can be dealt with at various times independently of the remainder for the purpose of retrieval, since the node relationships do not change with new arrivals. It is for this reason that discrete simulation priority queues are maintained within the random access memory by a sort-in from the bottom of the list.

To study the effects of dealing exclusively with the TFM priority queue, three cases are used. The a, b, c subcases of these three cases will be discussed later. Cases one, two and three differ only by the number of keys. They represent respectively the situations where a single primary key (PK) is used for retrieval, where a primary key is used with a secondary key (SK) to establish priority among equals, and finally (for the associative architecture) where a third key is added for the default ranking that is built into the RAM algorithms. The third key is not needed in the RAM because of the nature of the insertion algorithm working in conjunction with the RAM storage structure. The insertion algorithm involves the two-step process of first sorting the node into the proper position within the list and then arranging the proper linkage. By virtue of the sort-in step, ties in key values are automatically broken by a FIFO ranking. This is sometimes referred to as default ranking or stable sorting. This is not the case within the associative architecture where an extra key must be used to break ties. This extra key is used in case three.

The primary key (PK) is the main value used to operate the priority queues. In the case of time flow mechanisms it would be the simulation time. In some cases this is not adequate and a priority key is used in conjunction with and acts as a refinement of the primary key. This is indicated here as the secondary key (SK). Finally a tertiary key (TK) is used in the associative cases to implement the stable sort. This last key acts as another refinement of the primary key.

These are the three principal search techniques used in conjunction with cases one, two and three for the purpose of studying priority queues and time flow mechanisms. The first is the standard technique of searching for the minimum value based on a sort maintained list. This technique applies to the RAM. The second technique is Feng's $\gamma = 1$ minimum associative search referred to previously. This is used for the AM and AM/RAM architectures. The third technique is based on Lewin's algorithm and applies to the AM/RAM/AML architecture. The first two techniques are similar and are used to implement the variable time increment TFM either on the RAM, the AM or the AM/RAM.

The third technique, based on some experimentation and on the work of Morgan and Siegel [50], is an algorithm worked out to use Lewin's memory and algorithm effectively. This algorithm is called the fixed increment minimum value (FIMV) TFM. Lewin's algorithm is a subset of the FIMV time flow mechanism. The FIMV algorithm combines the fixed time increment and variable time increment techniques to yield a TFM that returns nodes in order, as in the VTI TFM, but has the speed of the FTI TFM without the errors mentioned earlier. The FIMV technique works in the following manner. A time increment Δt is established at the outset of the simulation. It is selected so that there is a high probability that there will be at least one potential state change occurring within Δt. This is the approach opposite to the normal way a time increment is selected for the FTI TFM [16]. The simulation clock register is incremented by the amount Δt and a less than or equal search is conducted. This is a parallel search as opposed to the minimum value search. If the search returns no nodes, the clock is again incremented. If one node is returned, the clock is set to the event time, that node is processed, and the clock is again incremented. Regardless of whether there are one (case one), two (case two) or three (case three) keys involved for the priority queue, the less than or equal search is only conducted on the primary key for the fixed increment part. If the search returns more than one node, then a transfer occurs of all returned nodes to the AML. This transfer includes all keys (one, two or three) and any links that are necessary. The keys are then stored in the AML horizontally from left to right (highest order to lowest order bit) instead of vertically by word. This is because Lewin's algorithm operates on all bits simultaneously. Lewin's algorithm is then initiated to return the nodes in order. In this way the fixed increment portion is a quick way to select the top independent nodes of the list, and Lewin's algorithm is a quick way to order just those nodes selected by the fixed increment and not all the nodes.

There are four outcomes for the fixed increment portion of the FIMV TFM algorithm. These are the return of a single node, the return of M identical nodes, the return of M dissimilar nodes, and the return of M nodes, some of which are identical. In this research the first three outcomes are considered for the purpose of collecting data. A problem arises in comparing the three selected outcomes for the FIMV TFM algorithm based on the AM/RAM/AML architecture with the algorithms used for the RAM, AM, and AM/RAM architectures. For this reason three subcases for each of the priority queues are introduced. These are subcases a, b and c respectively.

In subcase a the single node is the specified outcome. This corresponds to the development specified so far for a composite node cycle. That is, the measurement (discussed in the next section) is applied to a single composite node cycle whose output is the birth and death of a single node under parameterized conditions for PK (case one), PK/SK (case two), and PK/SK/TK (case three).

In subcase b the multiple identical nodes are the specified outcome. This corresponds in the RAM architecture to repeated trials of the composite node cycle. In the AM and AM/RAM architectures this corresponds to repeated trials of the composite node cycle with the exception that only a single search is required to support all M composite node cycles. This comes about because the match indicator in the associative memory indicates ties.

The ties are based on PK alone (case one), PK and SK (case two), and PK, SK, TK (case three). Subcase b or c is determined in the MV (minimum value) portion of the algorithm after transfer. For subcase b, Lewin's algorithm has the inherent ability to determine if all the entries are identical at the outset. Therefore in subcase b the FIMV algorithm immediately loads the first responder (lowest numbered memory location) into the appropriate register(s) and transfers control to the simulation control code (SCC). The SCC acts as the discrete simulation sequencer. Nodes are then removed in order as part of the composite node cycle.

Subcase c, M dissimilar nodes, is the one commonly occurring in discrete simulation. For the RAM, AM and AM/RAM, M repeated trials are required for the composite node cycles without any special savings realized by AM or AM/RAM architectures. In the case of the AM/RAM/AML architecture, a savings is realized in the MV portion of the FIMV algorithm, because Lewin's algorithm takes on the average two compare cycles per node. This is significantly different from the $\gamma = 1$ minimum search where computation time grows linearly with field width. After the outcome of the FIMV algorithm, control is again transferred to the SCC program which requests nodes individually as part of the composite node cycle.

In all outcomes of the FIMV algorithm, cases or subcases, the simulation clock is always set to the next event time. This is accomplished automatically in the SCC after node retrieval. This prevents the time disparity evident in the FTI algorithm. Therefore it is always applied to the correct time.

In Chapter IV the research results for cases one, two and three under priority queues are grouped according to subcase--that is, subcase a is presented first for each of the separate cases, then subcase b, and finally subcase c. In this manner the effect of the specified outcomes on each of the architectures can be presented and compared. Other situations that can occur in the AM/RAM/AML architecture are discussed in Appendix B.

Search Keys - Priority Queues, Cases 4, 5, 6

Cases four, five and six represent special cases of priority queues found in discrete simulation. Case four is find the first node in a list that meets the search criterion (equivalent to an FTI TFM), case five is find all nodes that meet the search criterion, and case six is find the minimum or maximum node. These three cases assume that the list is maintained in a random fashion in the RAM instead of in physical order for queues or ranked order for the first three cases of priority queues. This situation of having to search unordered lists comes about in discrete simulation when it is necessary to search a list that is ordered on a key other than the one which is to participate in a search, and it is either not felt to be worthwhile to maintain the list with two or more dissimilar keys or the ability to do so is not provided in the simulation language or package. These last three cases therefore require the extra search algorithm. All implementations are based on a single key search.

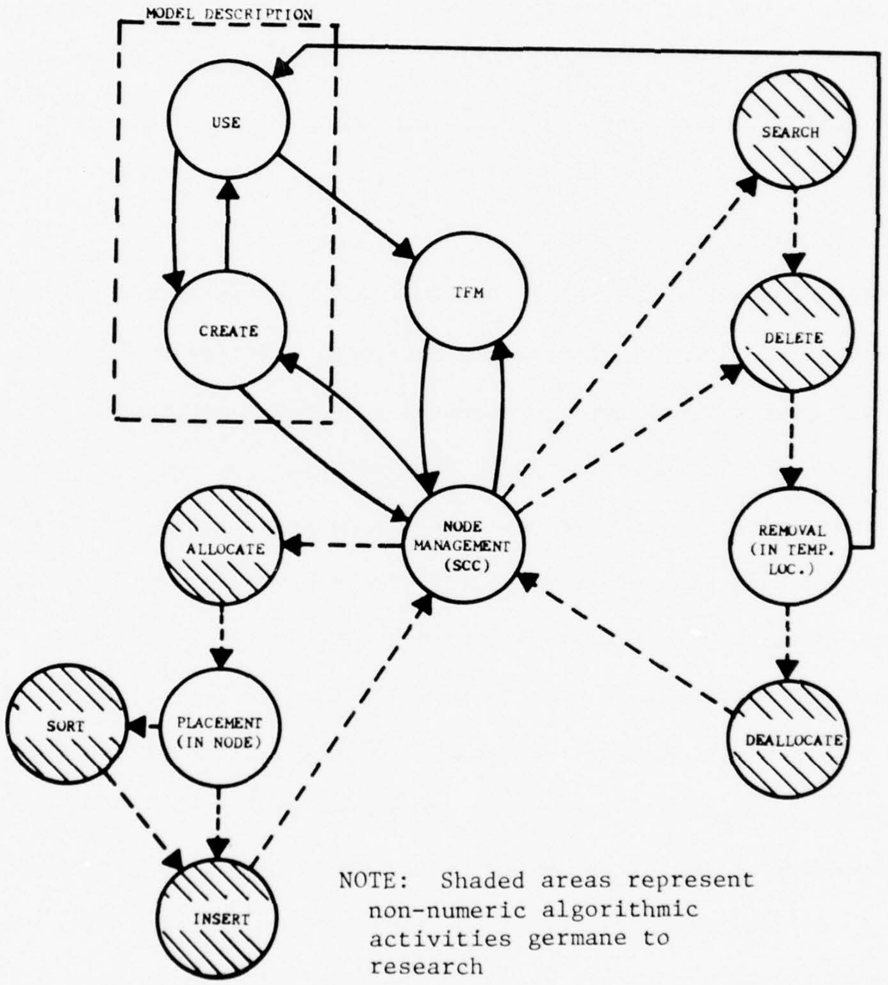
Parameters

The parameters necessary to characterize the various cases are shown in Figure 4. The first group of parameters, a, b, c, d, e and $\vec{l_i}$ pertain to RAM implementations. The first parameter, a, is the expected number of nodes that must be sorted through before the proper location of the node to be inserted can be found. This assumes no ties. The next parameter is the expected number of ties that must be resolved before the node to be inserted can be properly located. Parameter c represents the expected list depth in nodes before the first success. It pertains to case four under priority queues. Parameter d is the expected number of successful nodes in case five under priority queues. Parameter e represents the expected number of interchanges for selecting the maximum or minimum (case 6) and $\vec{l_i}$ represents the list length in nodes.

In the associative cases, the only parameter is $\vec{f}_{i,k}$, which represents the width in bits of the k^{th} field in the i^{th} composite node cycle (discussed below). In short, list length and node position are unimportant in the associative architectures used in this research.

Node Cycle

To study the behavior of the nodes under the conditions described in the previous section on algorithms again suggests a comparison to the birth and death process. To do this, the concept of a node cycle is introduced. The various steps in the birth and death process and node cycles are illustrated in Figure 6. At the



NOTE: Shaded areas represent non-numeric algorithmic activities germane to research

Figure 6. Discrete Simulation Node Cycles

center of the graph is node management, which manages the nodes by way of using the algorithms previously discussed. Requests for non-numeric processing can come either from the use/create cycle, which is typically part of the model description function, or directly through the time flow mechanism. A node cycle is then defined as the completion of a non-numeric path starting with and terminating on node management. Examples are allocate-placement-insert (birth) or search-delete-removal-deallocate (death). The removal step provides the node information to the use step.

Composite Node Cycle

The composite node cycle is made up by concatenating the birth and death cycles. In other words the total lifetime of the node in the information structure is measured for each memory organization and each information structure. This was done not only because of the many alternatives possible but because there are steps (algorithms) in the node cycle where one of the other memory organizations shows an advantage. Therefore a comparison at the cycle or step basis may be misleading in terms of overall performance. This may appear as an aggregate approach, which it partially is, but the research is so laid out that the individual steps causing poorer performance of one or the other memory organizations can be individually investigated as part of the extension to the research. The composite node cycles are shown in Appendix B in Tables B-1, B-2 and B-3 for each memory organization.

3.4. Measurements

The criterion used for comparison in this research is total computer model computation time for a composite node cycle. As will be seen in Chapter IV, this results in a parametric equation. This approach is in keeping with the approach used by Knuth to determine the usefulness of various algorithms. Alternative measures such as the number of compares, the number of memory instructions or the number of memory interrogations (without qualification) were considered and discarded as inappropriate for any of the following reasons:

1. Not all the algorithms encountered within a node cycle used compares.

2. During a node cycle in one memory certain algorithms would not use compares, but another memory would. The simple queues are an example.

3. During the node cycle, in some cases, a percentage of the work involved auxiliary and not memory instructions.

4. Memory interrogations (without qualification) are insensitive as a function of field width to the total amount of time needed by certain memories to complete a certain process.

5. Measures other than total time are insensitive to some or all of the following--the processing rates, processing widths (number of bits simultaneously active) and the transfer width (number of bits transferred simultaneously per unit of time).

6. Overhead is not considered.

7. Relative memory speeds are not differentiated.

The selection of total time then represents a superset of the previously considered measures. As such any of the other measures are derivable from the algorithms or from the composite node cycles.

The use of total time also permits a consideration of the relative speeds of the various memories through their instruction set timing. This allows the relative speed to be introduced as a parameter. The importance of this has been brought out by Thurber and Berg [68] and also by Weinberger [70] since total cost is tied closely to memory speed. In other words, if the results indicate that the associative memories must be much faster than the random access memories to achieve an overall processing advantage, there is reduced interest in studying them until technology reduces their cost [30].

CHAPTER IV

RESEARCH RESULTS

4.1. Introduction

The method used to portray the results of comparing the various architectures is based on the dichotomy that total model computation time for a composite node cycle for the RAM is based on the presence of other nodes within a list, while for the associative cases it is based on the bit width of keys and links. To exploit this dichotomy the concept of a breakeven bit width $\vec{E}_q$ is introduced. This is the value in terms of associative memory compares that is allocated to the combined search key and link fields used by the associative memory within a particular composite node cycle. To generate the $\vec{E}_q$ equations the total time taken by the RAM architecture is set equal to the total time (TT_i) taken by the AM, AM/RAM or AM/RAM/AML architecture for the ith composite node cycle. This concept can be formulated in terms of Equations 4-1a, 4-1b and 4-1c below.

MIX-RAM vs. MIX-AM	$TT_i(P_{RAM}) = TT_i(P_{AM})$	4-1a
MIX-RAM vs. MIX-AM/RAM	$TT_i(P_{RAM}) = TT_i(P_{AM/RAM})$	4-1b
MIX-RAM vs. MIX-AM/RAM/AML	$TT_i(P_{RAM}) = TT_i(P_{AM/RAM/AML})$	4-1c

P_{RAM}, P_{AM}, $P_{AM/RAM}$ and $P_{AM/RAM/AML}$ are respectively the parameters appropriate to each architecture. Equation 4-1a, 4-1b or 4-1c is a shorthand form of Equation 4-2. On the left hand side of Equation 4-2, Y_{i1} and Y_{i2} represent the RAM parameters, P_{RAM}, as coefficients of the two RAM computation times. The right hand side of Equation 4-2 is made up of two parts. The first, which is the linear summation of all associative compare times, will become $\vec{E}_q$. The second part consists of the Y_{i3} and Y_{i4} coefficients of the two associative computation times. They represent the P_{AM}, $P_{AM/RAM}$, or $P_{AM/RAM/AML}$ parameters, depending on whether Equation 4-2 corresponds to Equation 4-1a, 4-1b or 4-1c.

$$Y_{i1}\,T_M + Y_{i2}\,T_A = \sum_k \vec{f}_{i,k}\,T'_M + Y_{i3}\,T'_M + Y_{i4}\,T'_A \qquad 4\text{-}2$$

In Equation 4-2, i pertains to the ith composite node cycle and k to the associative fields used to participate in composite node cycle i.

All the graphs used to illustrate the results are based on Equation 4-2 with certain assumptions, which will follow, applied. The equivalent breakeven bit width ($\vec{E}_q$) is calculated as follows from Equation 4-2:

$$\sum_k \vec{f}_{i,k}\,T'_M = Y_{i1}\,T_M + Y_{i2}\,T_A - Y_{i3}\,T'_M - Y_{i4}\,T'_A \qquad 4\text{-}3a$$

$$\vec{E}_q = \sum_k \vec{f}_{i,k} = Y_{i1}\,\frac{T_M}{T'_M} + Y_{i2}\,\frac{T_A}{T'_M} - Y_{i3}\,\frac{T'_M}{T'_M} - Y_{i4}\,\frac{T'_A}{T'_M} \qquad 4\text{-}3b$$

$$\vec{E}_q = \sum_k \vec{f}_{i,k} = Y_{i1}\alpha + Y_{i2}\beta_1 - Y_{i4}\beta_2 - Y_{i3} \qquad 4\text{-}3c$$

where $\alpha = \dfrac{T_M}{T'_M}$, $\beta_1 = \dfrac{T_A}{T'_M}$, and $\beta_2 = \dfrac{T'_A}{T'_M}$.

In Equation 4-3b, the total search key and link field width in bits ($\sum_k \vec{f}_{i,k}$) for the associative architectures is set equal to a linear combination of the Y_{ij}'s and the computation times. In

Equation 4-3c, three additional parameters are introduced, alpha, beta_1 and beta_2, representing various timing ratios. Equation 4-3c is sufficiently complete to consider trade-off studies of architecture performance based on beta_1 and beta_2 for an equivalent breakeven bit width. In one respect Equation 4-3c is a primary result of this research in that it represents the culmination of one inclusive--although lengthy--approach to comparing various associative and sequential architectures by the two-step approach of specifying the architecture in terms of the instruction set and specifying the algorithm in terms of the storage structure and instruction set. Extended approaches based on $\vec{E}_q$ should yield an equation of the same form as Equation 4-3c with perhaps greater or fewer Y_{ij}'s.

Equation 4-3c is still too complex to yield simple graphic results. Therefore two assumptions are applied to the equation to yield the results, which are presented graphically later in the chapter.

The first assumption used for plotting is that $T_A = T'_A$; that is, that the time required to execute an auxiliary non-memory instruction such as increment, decrement, et cetera, is the same for all architectures. This assumption is a direct result of the manner in which the architectures were defined. Each architecture has the same registers and the same set of auxiliary instructions. As mentioned earlier this similarity was enforced to attempt within a comparable architectural framework to isolate the SISD approach from the SIMD approach. The result of the assumption is that beta_1 equals beta_2, and thus a single parameter beta may be used to yield Equation 4-4 from Equation 4-3.

$$\vec{E}_q = \sum_k \vec{f}_{i,k} = (Y_{i2} - Y_{i4})\beta + Y_{i1}\alpha - Y_{i3} \qquad 4\text{-}4$$

The mathematical comparison and the graphs are all based on Equation 4-4.

The second assumption used for the graphical presentation of the results is that beta is equal to 0.5. This is the value Knuth uses for his books, and a few years ago it did represent the average ratio of auxiliary instruction time to compare instruction time based on commercial implementation for RAM computers. Currently the trend seems to indicate beta should be closer to 1.0; however, for consistency with Knuth, 0.5 will be used, and discussion is provided for the general effect of beta.

Equation 4-4 does not yet represent the final form taken by the results. Two additional parameters, γ and δ, must be introduced in Equation 4-5.

$$\vec{E}_q = \sum_k \left\lceil \frac{\vec{f}_{S i,k}}{\gamma} \right. + \left\lceil \frac{\vec{f}_{\Gamma i,k}}{\delta} \right. = (Y_{i2} - Y_{i4})\beta + Y_{i1}\alpha - Y_{i3} \qquad 4\text{-}5$$

γ, as mentioned earlier, is a factor used to increase the degree of parallelism where possible during a search operation. For instance, suppose $\sum \vec{f}_{S i,k}$ is twenty bits where $\vec{f}_{S i,k}$ is the number of bits in search field k for a specified k. If the search is a fully parallel search such as a less than or equal, equal, greater than, et cetera, then γ may usefully be increased up to the limit of the field width, in this case twenty. A similar situation exists for δ in the term $\left\lceil \frac{\overrightarrow{f}_{T i,k}}{\delta} \right\rceil$ where $\overrightarrow{f}_{T i,k}$ is the transfer field width for field k. In essence δ measures the parallelism of the transfer mechanism. Equation 4-5 then represents the manner in which the results are presented.

The graphical results are plotted against alpha as the independent variable. This was based in part on the work of Thurber and Berg [68] and Weinberger [70], both of whom suggested that the memory portion of the computer would be the most critical, particularly where Thurber and Berg pointed out (and Hodges [30] supported) that from a cost point of view, the associative machine would have to have a slower memory, and hence slower compare times. Therefore the results are presented in such a way as to investigate this prior work to see if a slower associative memory would still be competitive from a total time measurement.

To illustrate the preceding discussion--and in particular the equivalent breakeven bit width--more clearly, the following example, listed in Table 4, is presented in detail. A few preliminary remarks are included below as a preamble to the example.

Figure 4 delineates fifty-seven composite node cycle and architectural situations. These situations are detailed in Appendix B as follows. Table B-1 specifies the total composite node time for each of the twelve situations for queues. This is the total amount

Algorithms	RAM		AM	
	Time	Composite Node Cycle	Time	Composite Node Cycle
Allocate	$4T_M + 2T_A$	A_1	$3T'_M + 3T'_A$	A'_1
Insert	$6T_M + T_A$	I_1	$3T'_M + 2T'_A$	I'_2
Delete	$7T_M + 2T_A$	D_1	$(1 + \left\lceil \overrightarrow{LN}/\gamma + \overrightarrow{PK}\right.)T'_M + 3T'_A$	D'_3
Deallocate	$3T_M + T_A$	DA_1	T'_M	DA'_1
Total Time	$20T_M + 6T_A$		$(8 + \left\lceil \overrightarrow{LN}/\gamma + \overrightarrow{PK}\right.)T'_M + 8T'_A$	

Equation 4-1a

$TT_i(P_{RAM}) = TT_i(P_{AM})$ $\qquad$ $20T_M + 6T_A = (8 + \left\lceil \overrightarrow{LN}/\gamma + \overrightarrow{PK}\right.)T'_M + 8T'_A$

Equation 4-2

$Y_{i1}T_M + Y_{i2}T_A = \overrightarrow{f}_{i,k}T'_M + Y_{i3}T'_M + Y_{i4}T'_A$ $\qquad$ $20T_M + 6T_A = (\left\lceil \frac{\overrightarrow{LN}}{\gamma} + \overrightarrow{PK}\right.)T'_M + 8T'_M + 8T'_A$

Equation 4-3a

$\Sigma\overrightarrow{f}_{i,k}T'_M = Y_{i1}T_M + Y_{i2}T_A - Y_{i3}T'_M - Y_{i4}T'_A$ $\qquad$ $(\left\lceil \frac{\overrightarrow{LN}}{\gamma} + \overrightarrow{PK}\right.)T'_M = 20T_M + 6T_A - 8T'_M - 8T'_A$

Equation 4-3b

$\overrightarrow{E}_q = \Sigma\overrightarrow{f}_{i,k} = Y_{i1}\frac{T_M}{T'_M} + Y_{i2}\frac{T_A}{T'_M} - Y_{i3} - Y_{i4}\frac{T'_A}{T'_M}$ $\qquad$ $\left\lceil \frac{\overrightarrow{LN}}{\gamma} + \overrightarrow{PK}\right. = 20\frac{T_M}{T'_M} + 6\frac{T_A}{T'_M} - 8 - 8\frac{T'_A}{T'_M}$

Equation 4-3c

$\overrightarrow{E}_q = \Sigma\overrightarrow{f}_{i,k} = Y_{i1}\alpha + Y_{i2}\beta_1 - Y_{i3} - Y_{i4}\beta_2$ $\qquad$ $\left\lceil \frac{\overrightarrow{LN}}{\gamma} + \overrightarrow{PK}\right. = 20\alpha + 6\beta_1 - 8 - 8\beta_2$

Equation 4-4

$\overrightarrow{E}_q = \Sigma\overrightarrow{f}_{i,k} = Y_{i1}\alpha + (Y_{i2} - Y_{i4})\beta - Y_{i3}$ $\qquad$ $\left\lceil \frac{\overrightarrow{LN}}{\gamma} + \overrightarrow{PK}\right. = 20\alpha - 2\beta - 8$

Equation 4-5

$\overrightarrow{E}_q = \Sigma \left\lceil \frac{\overrightarrow{f}_{si,k}}{\gamma}\right. + \left\lceil \frac{\overrightarrow{f}_{ri,k}}{\gamma}\right.$ $\qquad$ $\left\lceil \frac{\overrightarrow{LN}}{\gamma}\right. + \overrightarrow{PK} = 20\alpha - 2\beta - 8$

$= (Y_{i2} - Y_{i4})\beta + Y_{i1}\alpha - Y_{i3}$

Numerical values for $\alpha = 1$, $\beta = 0.5$ $\qquad$ $\left\lceil \frac{\overrightarrow{LN}}{\gamma} + \overrightarrow{PK}\right. = 11$

Table 4. Example of $\overrightarrow{E}_q$ Equation Development

of computational time (in parametric form) necessary to complete the composite node cycle listed to the right of the table in abbreviation. Similarly, the thirty-six situations formed from cases one, two and three of priority queues along with their sub-cases are specified in Table B-2, with the remaining nine situations for priority queues four, five and six in Table B-3. The total computational time listed in each of these tables is computed by first listing each algorithm participating in a particular composite node cycle and then obtaining from Table B-4 the total computational time for that algorithm. The total composite node cycle time is then the summation of the separate algorithm times. Table B-4 also lists the figure number in Appendix B that shows the individual algorithmic flow chart. It is possible then by way of Equations 4-1 through 4-5 and Appendix B to study how an algorithm or subgroup of instructions affects the outcome of a particular comparison.

To return to the specific example, the top of Table 4 reproduces the entries in Table B-4 appropriate to case one of queues and in particular to the situation comparing the RAM to the AM. The specific algorithms (A_1, I_1, D_1 and DA_1 for the RAM and A_1', I_2', D_3' and DA_1' for the AM) forming a composite node cycle are taken from Table B-1. The total time for each architecture is then summed in Table 4.

The remaining portion of Table 4 develops a side-by-side comparison of Equations 4-1 through 4-5 with the actual numerical example for queue case one. In Equation 4-1 the total time for the RAM is set equal to the total time for the AM. This is the key step in developing the equivalent breakeven bit width concept. By setting the total times equal, the equality is maintained or destroyed by the values of the α, β, γ, ς and $\Sigma\overrightarrow{f_{i,k}}$ parameters. The first four relate directly to the hardware in terms of relative processing times, or the degree of parallelism apparent in the associative architectures. As such they eventually relate to hardware cost. The last parameter is the equivalent breakeven bit width, which is permitted to vary as the eventual balancing factor to maintain equality. This balancing process evolves from the rest of the example.

In Equation 4-2 in Table 4 the two times are set equal in expanded form. Equations 4-3a, 3b and 3c partition the $\vec{E}_q$ term and set up the parametric ratios. Equation 4-4 introduces the assumption that $T_A = T_A'$, hence $\beta_1 = \beta_2 = \beta$. Equation 4-5 partitions the $\overrightarrow{E_q}$ term into the memory contribution $\left|\dfrac{\overrightarrow{f_{S_{i,k}}}}{\gamma}\right.$ and the transfer contribution $\left|\dfrac{\overrightarrow{f_{T_{i,k}}}}{\varsigma}\right.$. Note in this example the $\vec{E}_q$ does not involve a transfer term, and, further, the coefficient of α is non-parametric. This is only true because of the simpler nature of queues, discussed in Chapter III.

The final step results from substituting $\alpha = 1$ and $\beta = 0.5$ into the equation. Alpha equal to one means that the time to complete a full word compare in the RAM is equal to the time required to make a bit slice compare in the AM. The result is that to balance the equation, $\overrightarrow{E}_q$ must equal eleven. The question is, eleven what? The term bit width or more properly equivalent bit width (EBW) is introduced to answer this question. This is the number of bits necessary to represent the maximum values of the fields necessary to implement a particular composite node cycle on any of the associative architectures. Herein lies the dichotomy between the RAM and the associative architecture--that is, the total processing time of the latter is based on how many bit slices or equivalent bit widths can be processed in parallel,[3] while the former depends on efficient node organization. The term equivalent bit width is used because of the potential parallelism of the associative architecture. As discussed previously the associative architecture may process (search or transfer) in parallel (simultaneously) from one to M_w bits. This is a function of the internal logic. In the specific example just cited $\overrightarrow{E}_q$ was made up of two terms, $\left\lceil \frac{\overrightarrow{LN}}{\gamma} \right.$ and $\overrightarrow{PK}$. The former term represents a fully parallel search where the number of bits participating within each compare cycle is a function of γ . The search on PK is done by Feng's bit slice search, which is fixed by internal logic at one bit per compare. Thus γ is always one and hence omitted from the equation since including it would give an improper connotation. Therefore consider a $\overrightarrow{PK}$ of ten bits. This means ten of the eleven bit widths available to balance the equation have already been used up. This is equivalent $\overrightarrow{PK}$ value of $2^{10} - 1$ or 1023. In terms of queue length, this means 1023 nodes within each queue list can be accommodated.

[3]The associative parallelism is also affected by insufficient memory size to contain the total searched list. However, in this research the memory length is considered adequate.

If only straight bit widths or slices were concerned, there would be one remaining to balance the equation. However, by increasing γ from one to M_w, up to $2^{M_w}-1$ bits or lists can be accommodated. Equivalent bit widths then come as a result of modifying gamma, which is the same as modifying the degree of parallelism within the associative architecture. Delta plays a similar role for data transfer involving Lewin's algorithm and memory.

To complete the example $\vec{E}_q$ is now discussed. The formulation discussed up to now in Chapter IV has concentrated on defining and explaining $\vec{E}_q$. The question as to whether the RAM or one of the associative architectures is superior ultimately depends on whether the $\vec{E}_q$ required by the particular composite node cycle is less than (associative superiority), equal to (a draw), or greater than (RAM superiority) the $\vec{E}_q$ dictated by fixing α, β, γ and δ.

In each case where the number of EBW's is either less than or greater than the number required for equation balancing, the discrepancy converts directly to associative compare times T'_M. In terms of the example, if PK were five bits and $\left\lceil \frac{\overrightarrow{LN}}{\gamma} \right.$ were one bit, then each time the composite node cycle was executed $5T'_M$ would be saved in actual running time. Conversely, if $\overline{PK}$ were ten bits and $\left\lceil \frac{\overrightarrow{LN}}{\gamma} \right.$ were five bits, then $4T'_M$ would be lost vis-a-vis the RAM for each composite node cycle. This latter situation could arise for a $\gamma = 1$ associative machine such as STARAN. The graphical results that follow represent a rapid method to ascertain $\overrightarrow{E_q}$ based on parameter fixing, or conversely, given $\overrightarrow{E_q}$ for a particular problem, to determine the hardware parameters for an appropriate associative implementation.

In the graphs, the lines represent a fixing of the β, γ, δ and other algorithmic parameters plotted against α. The left hand scale is then in EBW's. Therefore for a given α, $\overrightarrow{E_q}$ for equation balancing is read off the vertical axis. The line for the current example is the upper line shown in Figure 7. To determine the actual $\overrightarrow{E_q}$ required by the particular composite node cycle requires an analysis similar to that done for $\overline{LN}$ and $\overrightarrow{PK}$ in the example. This analysis is done in the context of discrete simulation processing requirements. For instance, in most discrete simulations one would expect priority queue list lengths to be less than one thousand and their number to be perhaps twenty or less. There may be other situations where list lengths might vary widely from these figures and would have to be considered accordingly.

As a final introductory note, two items should be kept in mind. First, the results to follow are based on stated assumptions. As such the conclusions drawn from them must not be generalized without due care beyond the assumptions. Second, the results are not by themselves intended to prove that one architecture is better than another. Instead they indicate what is required to make such a decision and, where the research assumptions are met and field widths and other parameters are known, permit such a decision to be made.

These decisions are shown as decision areas I and II on the graphs of the following results. Region I is favorable to the RAM, while region II is favorable to the particular associative architecture depicted by the graph. Each graph is labelled with a unique equation number that is also listed in the table of $\overrightarrow{E_q}$ equations appropriate to the particular results section.

4.2. Queues

The first information structures to be discussed in the results are the queues, the equations for which are shown in Table 5. These information structures operate in an unusual way both naturally and within the random access memory. This is because a queue represents a physical ordering of individual items. In the random access memory this becomes a physical ordering of nodes, which in turn implies that no keys are necessary because there is no searching for the next item in the queue. The next item in the queue is either the last item put in (LIFO) or the first remaining item left in the queue (FIFO).

This physical ordering represents a problem in implementing a queue within an associative memory because the memory is basically a parallel search device that selects nodes based on keys. Therefore it was necessary to convert the queue information structure into a key search structure. This was done in two

Information Structure	Comparison	Equivalent Breakeven Bit Width Equations	Equation Number
Queue (FIFO)	RAM vs. AM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} = 20\alpha - 2\beta - 8$	1
1	RAM vs. AM/RAM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} = 20\alpha - 2\beta - 10$	2
Queue (FIFO)	RAM vs. AM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \left\lceil \frac{\vec{C}_j}{\gamma} \right\rceil = 20\alpha - 4\beta - 12$	3
2	RAM vs. AM/RAM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \left\lceil \frac{\vec{C}_j}{\gamma} \right\rceil = 20\alpha - 4\beta - 14$	4
Queue (LIFO)	RAM vs. AM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} = 20\alpha - 2\beta - 8$	5
3	RAM vs. AM/RAM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} = 20\alpha - 2\beta - 10$	6
Queue (LIFO)	RAM vs. AM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \left\lceil \frac{\vec{C}_j}{\gamma} \right\rceil = 20\alpha - 4\beta - 12$	7
4	RAM vs. AM/RAM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \left\lceil \frac{\vec{C}_j}{\gamma} \right\rceil = 20\alpha - 4\beta - 14$	8

Table 5. $\vec{E}_q$ Equations for Queues

different ways, as indicated in Table 3. The first way was simply to insert a time of entry key in the node (Equations 1, 2, 5 and 6 in Table 5). This is straightforward in a discrete simulation since there is a simulation clock available. Based on such a key a LIFO queue can be maintained with a maximum search and a FIFO queue can be maintained with a minimum search. Method one then converts the queue into a priority queue, as will method two, discussed next.

The second method was to use either one counter for LIFO queues or two counters for FIFO queues (Equations 3, 4, 7 and 8 in Table 5). These methods were based on the fact that since queues are maintained in physical order it is possible uniquely to serial number the nodes in a contiguous fashion. Therefore in the case of LIFO queues, when the time came to remove the next node, an 'equal to' search on the last serial number stored selected the proper node. In the case of FIFO queues, the first counter was used to insert the serial number and a second was used for 'equal to' searches for removal. Two benefits accrued to this second method of maintaining queues in an associative memory. The first was that an 'equal to' search is a completely data parallel search. The second was that when queues are used in discrete simulation it is frequently required as part of the data generation function to know at various time points the number of items remaining in the queue. This information is automatically available in the LIFO case and is the difference of the two counters in the FIFO case.

Figure 7 and 8 illustrate graphically the results of comparing the two associative approaches with the single random access approach. Note that the equations for LIFO and FIFO queues are the same; therefore only one set is plotted.

The most important result is the role of gamma. To discuss this, consider again the previous numerical example, assuming an operating point of alpha equal to one. This means for Equation 1 in Table 5 that there are eleven memory cycles available for E_q. These memory cycles must be apportioned between two field terms $\left\lceil \frac{\overrightarrow{LN}}{\gamma} \right.$ and $\overrightarrow{PK}$. Recall that method one uses minimum or maximum searches. These searches are not fully data parallel because there must be some reconciliation among the words themselves as opposed to independent comparison with some external value. Feng has shown that such a search can be conducted with gamma equal to one [21]. To increase gamma requires either going to special circuitry for ordered retrieval, such as Lewin's algorithm, or the conversion from a minimum or maximum search to a fully data parallel search. To return to the example, Equation 1 in Table 5 reflects that the selection of the proper queue by the LN key exhibits full data parallelism, since each node is checked independently, which means gamma can be increased beyond one. If, however, gamma is one, and if there are some thirty-two linear lists colocated in the AM, there are six bits or EBW left for $\overrightarrow{PK}$, which may or may not be an unacceptable number for queue membership (63 members). If gamma were greater than or equal to $\overrightarrow{LN}$, ten EBW's would normally be adequate.

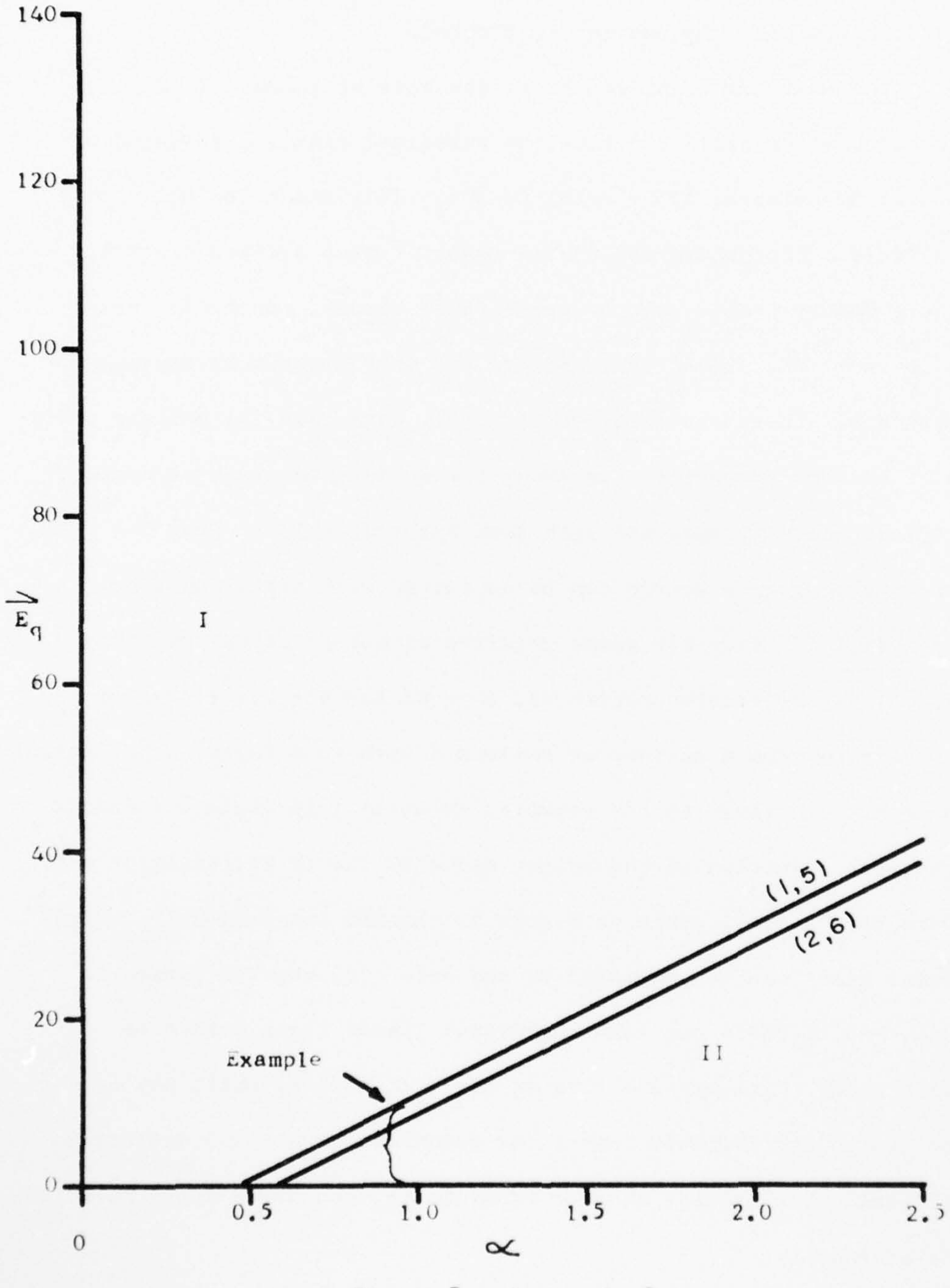


Figure 7. Queues 1, 3

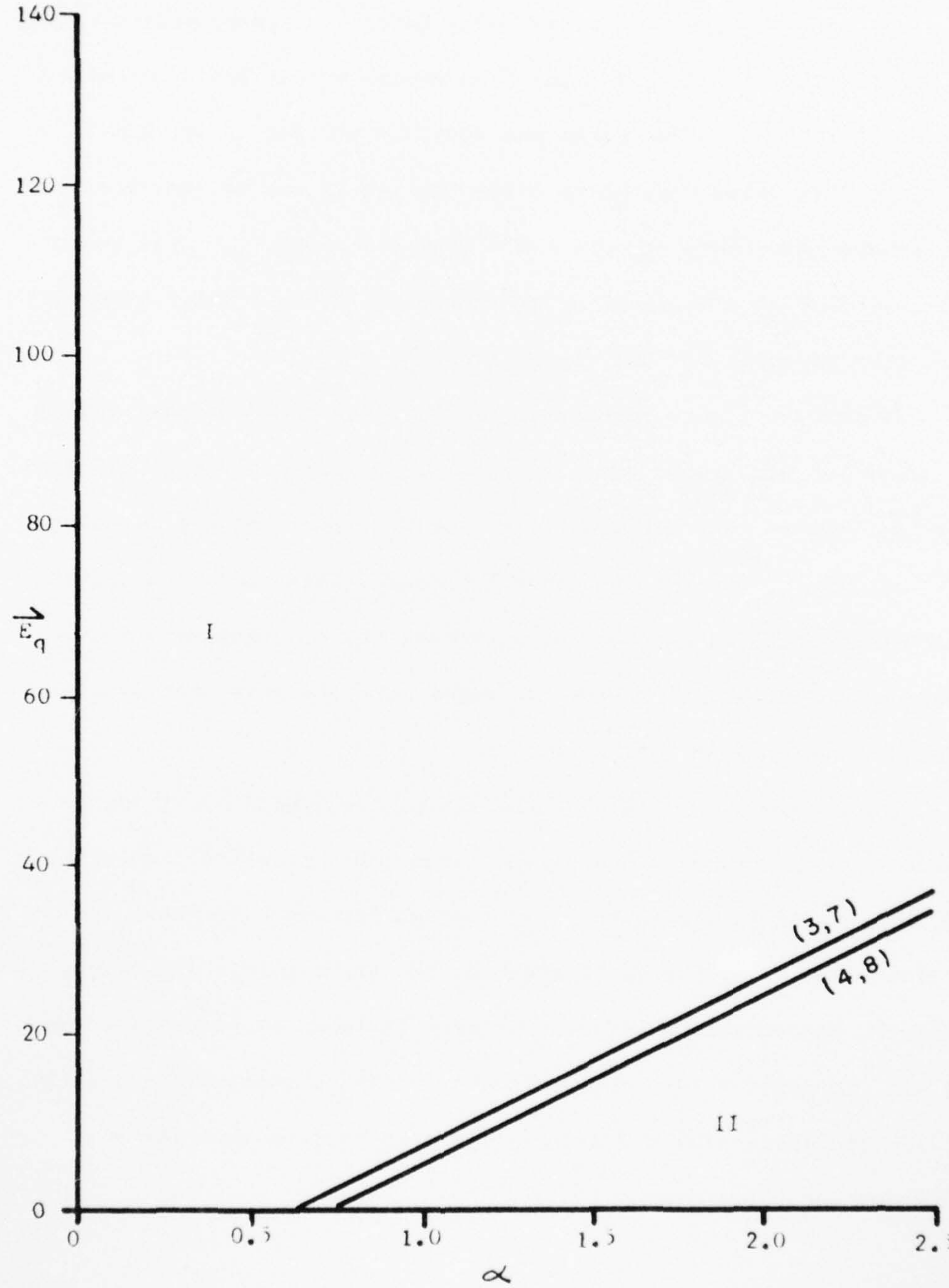


Figure 8. Queues 2, 4

However, method two--the counter method--permits gamma to play a role for both $\overrightarrow{LN}$ and $\overrightarrow{PK}$. Based on Figure 8 (Equations 3 and 7) at alpha equal to one, there are seven EBW's. If gamma were equal to the maximum of $\overrightarrow{LN}$ and $\overrightarrow{PK}$, only two memory cycles would be needed. Assume for a moment that gamma was equal to six and $\overline{LN}$ was equal to six. This means that up to sixty-four lists can be stored in the AM and that there are six EBW's left for $\lceil \overrightarrow{PK}/\gamma$. This means $\overrightarrow{PK}$ could take on a value of up to thirty-six bits, a value generally more than adequate for the counter values.

As perhaps a more interesting example consider the same values as above but with gamma equal to two. Three EBW's would be required for $\overline{LN}$, leaving four EBW's or eight for $\overrightarrow{PK}$. This means a value of 2^8-1 or 255 for each of the queue counters. This means that if no queue ever exceeded that number of nodes, the AM using method two would be on par with the RAM. If gamma or alpha were increased, the AM would show an advantage.

The second result is that the use of the hybrid architecture (AM/RAM) requires only an additional two EBW's to operate over the AM. The third result is that the timings for the algorithms are dominated by $Y_{i,1}$, the coefficient of the alpha term. This means that the preponderance of computer time is taken by memory instructions. In general by changing the algorithm to incorporate parallel search and increasing γ , associative architecture materially improves queue performance.

4.3. Priority Queues, Cases 1, 2, 3

The prime interest in studying priority queues within the context of discrete simulation is that they are the information structure used to implement time flow mechanisms based on the VTI method. Unfortunately, the priority queue does not seem to be well suited to the two primary associative architectures considered for this research, the AM and AM/RAM. This problem arises from several sources which will be discussed prior to presenting the results.

The priority queue is based on selecting the node with the greatest or smallest key value for the search field. For time flow mechanisms, the primary key is the simulation time, which means a search on the smallest time to determine the next potential state change. This means a minimum search, which as pointed out in the preceding section is not a fully data parallel search. Unlike the queue which by its nature permits a straightforward conversion to a full data parallelism by the use of counters, in the priority queue there is no guarantee that the nodes will be contiguous based on search key values. Two additional sources of difficulty arise. One is that the CDLLL is an efficient structure for representing the priority queue in the RAM when it is coupled with a sort-in process. Studies by Conway [8], Lave [44], Morgan and Siegel [50] and Knuth [36] indicate that in general it becomes more efficient with respect to its own overhead as the list grows and as the future state changes become less dense compared to a random list. The latter case of decreasing density seems to be the predominant case in discrete simulation. The last source of difficulty is that the sort-in method used for priority queues for RAM architecture has built within it a queue. This comes about because the sort-in process forces nodes with equal keys to be separated by a FIFO queue discipline. This last situation brings the dichotomy between the AM and the RAM into sharp relief because the AM composite node cycle time increases linearly with search field width and as was pointed out in the previous section, the AM requires an extra or artificial key to replace the physical queue order. This means that the AM to compare favorably must be able to absorb the extra field width, and compete with an efficient RAM process, without the benefit of a fully data parallel search technique, at least using the AM and AM/RAM architectures.

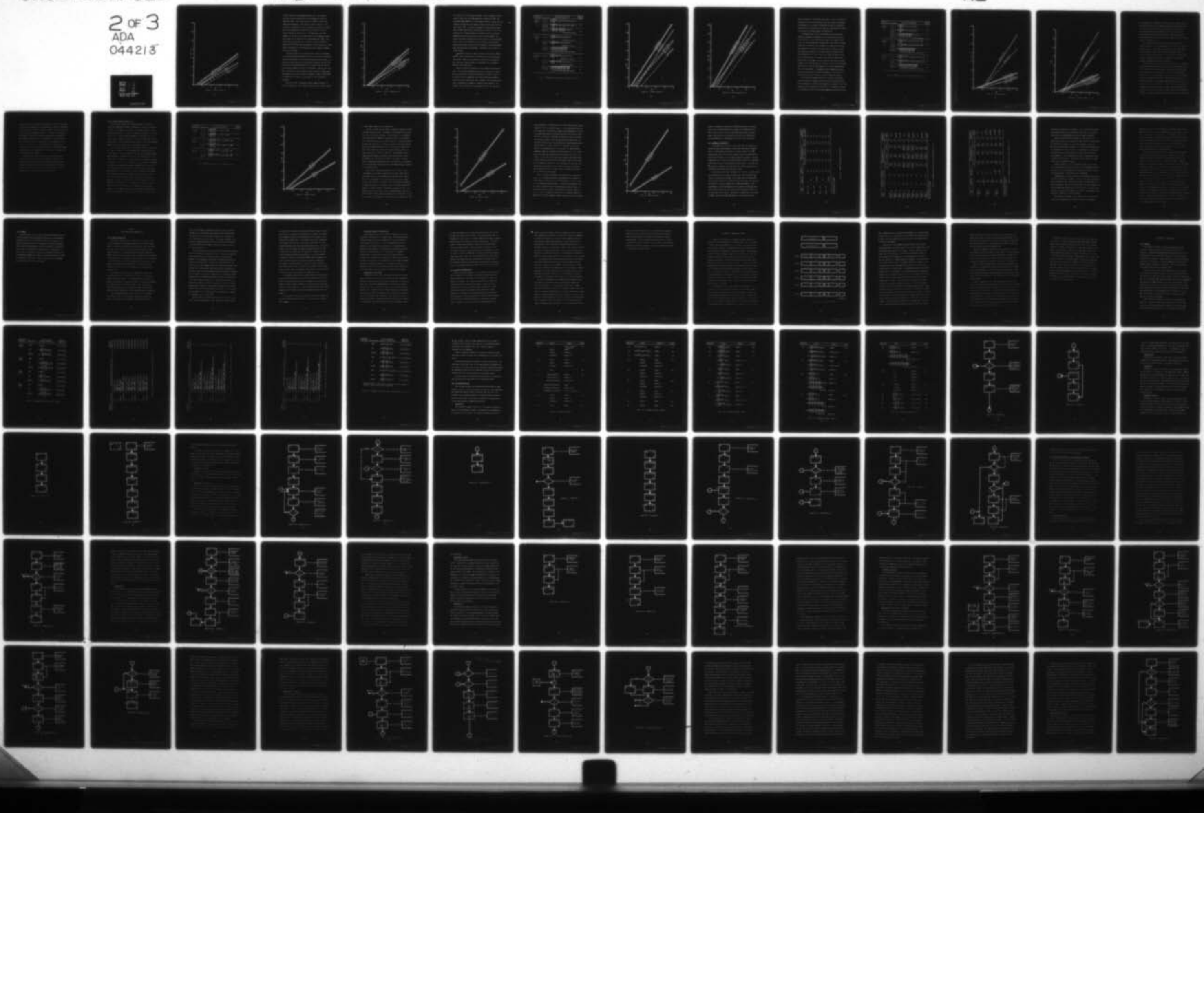
The results are presented in three major categories. The first includes 1a, 2a and 3a in Table 6. These results represent a composite node cycle where a single node is selected which corresponds to the next potential state change. The first group of equations (case 1a) in this category (9, 10, 11) is shown graphically in Figure 9 for the case where the parameter a is one and five. The parameter a is a sort-in factor, which is the number of nodes expected to precede the new node to be inserted into the priority queue before the new node is placed in its proper ranked order. The results indicate that as the parameter a increases, the value of $\overline{E}_q$ increases. The value of $\overline{E}_q$ is probably adequate at a = 5 and alpha = 1 for most simulations based on the GASP II

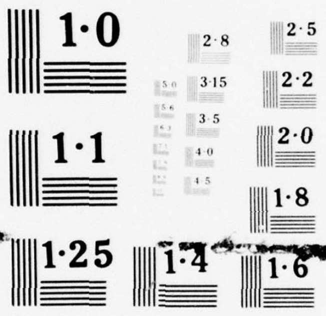
Information Structure	Comparison	Equivalent Breakeven Bit Width Equations	Equation Number
Priority Queue 1a	RAM vs. AM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \vec{PK} = (2a+21)\alpha + (a-1)\beta - 7$	9
	RAM vs. AM/RAM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \vec{PK} = (2a+21)\alpha + (a-1)\beta - 9$	10
	RAM vs. AM/RAM/AML	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \sqrt{\frac{\vec{PK}}{\gamma}} = (2a+21)\alpha + (a-7)\beta - 10$	11
Priority Queue 2a	RAM vs. AM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \vec{PK} + \vec{SK} = (2a+3b+22)\alpha + (a+3b-1)\beta - 7$	12
	RAM vs. AM/RAM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \vec{PK} + \vec{SK} = (2a+3b+22)\alpha + (a+3b-1)\beta - 9$	13
	RAM vs. AM/RAM/AML	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \sqrt{\frac{\vec{PK}}{\gamma}} = (2a+3b+22)\alpha + (a+3b-6)\beta - 10$	14
Priority Queue 3a	RAM vs. AM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \vec{PK} + \vec{SK} + \vec{TK} = (2a+3b+22)\alpha + (a+3b-2)\beta - 7$	15
	RAM vs. AM/RAM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \vec{PK} + \vec{SK} + \vec{TK} = (2a+3b+22)\alpha + (a+3b-2)\beta - 9$	16
	RAM vs. AM/RAM/AML	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \sqrt{\frac{\vec{PK}}{\gamma}} = (2a+3b+22)\alpha + (a+3b-6)\beta - 10$	17

Table 6. $\vec{E}_q$ Equations for Priority Queues 1a, 2a, 3a

AD-A044 213 ROME AIR DEVELOPMENT CENTER GRIFFISS AFB N Y F/G 9/2
A COMPARISON OF SEQUENTIAL AND ASSOCIATIVE COMPUTING OF PRIORIT--ETC(U)
AUG 77 B M LANDSON
UNCLASSIFIED RADC-TR-77-221 NL

2 OF 3
ADA
044213





NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

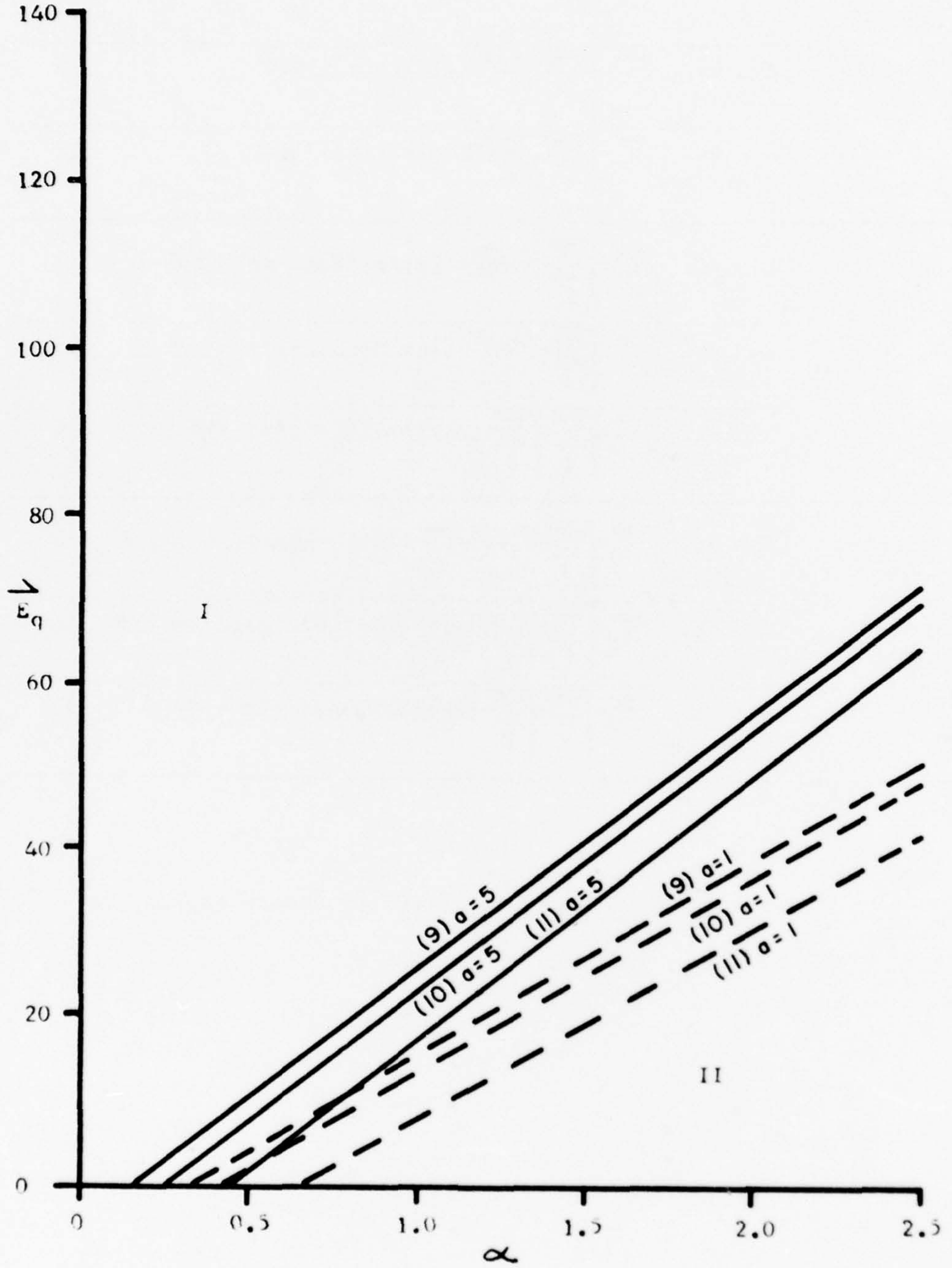


Figure 9. Priority Queue 1a

discrete simulation programming package [62]. This package initially allotted sixteen bits to the primary key, which is simulation time. This leaves ten bits for $\overrightarrow{LN}$ in the AM case, gamma notwithstanding. The problem is that the AM case is not necessarily representative since a separate key which is a refinement on the primary key has not been set aside for the FIFO default ranking in case of ties. This additional key would normally be the time of list entry based on the simulation clock, which means that it would be of the same magnitude as $\overrightarrow{PK}$. This would require thirty-two bits, then, with sixteen for $\overrightarrow{PK}$ and sixteen for $\overrightarrow{TK}$. $\overrightarrow{TK}$ is used to designate the FIFO queue key. Under these circumstances, the AM probably would not compare favorably, much less the AM/RAM.

The AML was added to the architectural choices to determine if a special searching capability--in this case Lewin's algorithm--would alleviate some of the difficulties mentioned previously. In general the AML architecture requires more overhead than the other associative cases, which means that the graphs shown in Figure 9 are displaced downward from their counterparts. However, note that $\overrightarrow{E_q}$ is now based on a fully parallel search procedure, which means that gamma can be used to reduce the number of EBW's for $\overrightarrow{LN}$ and $\overrightarrow{PK}$ (see Equation 11 in Table 6), resulting in a net associative advantage.

Case 2a in Table 6 represents another common situation in discrete simulation. That is where an extra field is used to break

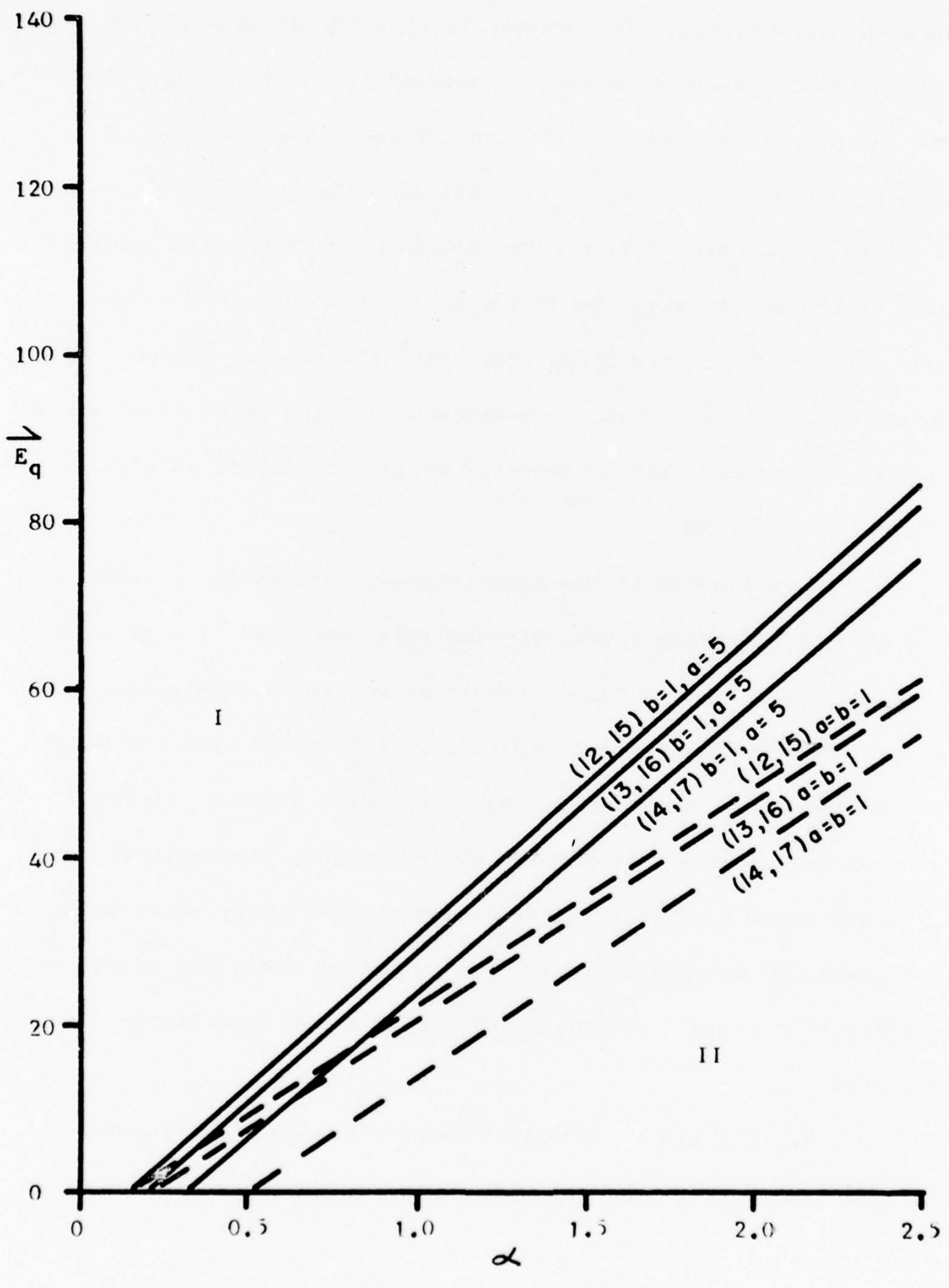


Figure 10. Priority Queue 2a, 3a

ties prior to the FIFO default ranking. This is normally called a priority field and in the $\overrightarrow{E_q}$ equations is indicated by $\overrightarrow{SK}$. An additional RAM parameter, b, also appears, which is another sort-in parameter dealing with the expected number of nodes that have the same value of $\overrightarrow{PK}$. It counts the number of nodes having the same value of $\overrightarrow{PK}$ through which a new node must be sorted. Case 3a has effectively the same equations as 2a except that for the AM and AM/RAM architectures the default field TK has been added. The results are shown in Figure 10 and they indicate, as was mentioned before, that as time increases with field width, coupled with a search method which is not fully data parallel, the AM and AM/RAM come into increasing disadvantage. Increasing b also favors the AM and AM/RAM.

Equations 14 and 17 in Table 6 reflect only two keys, $\overrightarrow{LN}$ and and $\overrightarrow{PK}$. This is because the FTI portion of the FIMV TM operates only on $\overrightarrow{PK}$. Figure 10 illustrates the results for Equations 14 and 17 and once again indicates the important role of gamma in making the AM competitive.

Multiple identical responses or state changes are considered next. The equations representing this case are listed in Table 7 and the results are illustrated graphically in Figure 11 and 12. The RAM 'a' and 'b' parameters have been replaced with expected values (over the M retrievals) and a parameter 'M' has been introduced to represent the number of identical state changes. The transfer width parameter delta now appears in the AML equations

Information Structure	Comparison	Equivalent Breakeven Bit Width Equations	Equation Number
Priority Queue	RAM vs. AM	$\overrightarrow{E}_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \overrightarrow{PK} = (2\bar{a}M+21M)\alpha + (M\bar{a}+2M-3)\beta - 6M - 1$	18
1b	RAM vs. AM/RAM	$\overrightarrow{E}_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \overrightarrow{PK} = (2\bar{a}M+21M)\alpha + (M\bar{a}+2M-3)\beta - 7M - 2$	19
	RAM vs. AM/RAM/AML	$\overrightarrow{E}_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \left\lceil \frac{\overrightarrow{PK}}{\gamma} \right\rceil + \left\lceil \frac{\overrightarrow{PK}}{\delta} \right\rceil + \left\lceil \frac{\overrightarrow{AMA}}{\delta} \right\rceil = M(2\bar{a}+21)\alpha$ $+ (M\bar{a}-M-10)\beta - 10M - 4$	20
Priority Queue	RAM vs. AM	$\overrightarrow{E}_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \overrightarrow{PK} + \overrightarrow{SK} = (2\bar{a}M+3M\bar{b}+22M)\alpha +$ $(M\bar{a}+3M\bar{b}+3M-4)\beta - 6M - 1$	21
2b	RAM vs. AM/RAM	$\overrightarrow{E}_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \overrightarrow{PK} + \overrightarrow{SK} = (2\bar{a}M+3M\bar{b}+22M)\alpha +$ $(M\bar{a}+3M\bar{b}+3M-4)\beta - 7M - 2$	22
	RAM vs. AM/RAM/AML	$E_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \left\lceil \frac{\overrightarrow{PK}}{\gamma} \right\rceil + \left\lceil \frac{\overrightarrow{PK}}{\delta} \right\rceil + \left\lceil \frac{\overrightarrow{SK}}{\delta} \right\rceil + \left\lceil \frac{\overrightarrow{AMA}}{\delta} \right\rceil =$ $\left[M(2\bar{a}+3\bar{b}+22)\alpha + (M\bar{a}+3M\bar{b}-10)\beta - 10M - 4 \right]$	23
Priority Queue	RAM vs. AM	$\overrightarrow{E}_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \overrightarrow{PK} + \overrightarrow{SK} + \overrightarrow{TK} = (2\bar{a}M+3M\bar{b}+22M)\alpha$ $+ (M\bar{a}+3M\bar{b}+3M-5)\beta - 6M - 2$	24
3b	RAM vs. AM/RAM	$\overrightarrow{E}_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \overrightarrow{PK} + \overrightarrow{SK} + \overrightarrow{TK} = (2\bar{a}M+3M\bar{b}+22M)\alpha$ $+ (M\bar{a}+3M\bar{b}+3M-5)\beta - 7M - 2$	25
	RAM vs. AM/RAM/AML	$\overrightarrow{E}_q = \left\lceil \frac{\overrightarrow{LN}}{\gamma} \right\rceil + \left\lceil \frac{\overrightarrow{PK}}{\gamma} \right\rceil + \left\lceil \frac{\overrightarrow{PK}}{\delta} \right\rceil + \left\lceil \frac{\overrightarrow{SK}}{\delta} \right\rceil + \left\lceil \frac{\overrightarrow{TK}}{\delta} \right\rceil + \left\lceil \frac{\overrightarrow{AMA}}{\delta} \right\rceil =$ $\left[M(2\bar{a}+3\bar{b}+22)\alpha + (M\bar{a}+3M\bar{b}-10)\beta - 10M - 4 \right]$	26

Table 7. $\overrightarrow{E}_q$ Equations for Priority Queues 1b, 2b, 3b

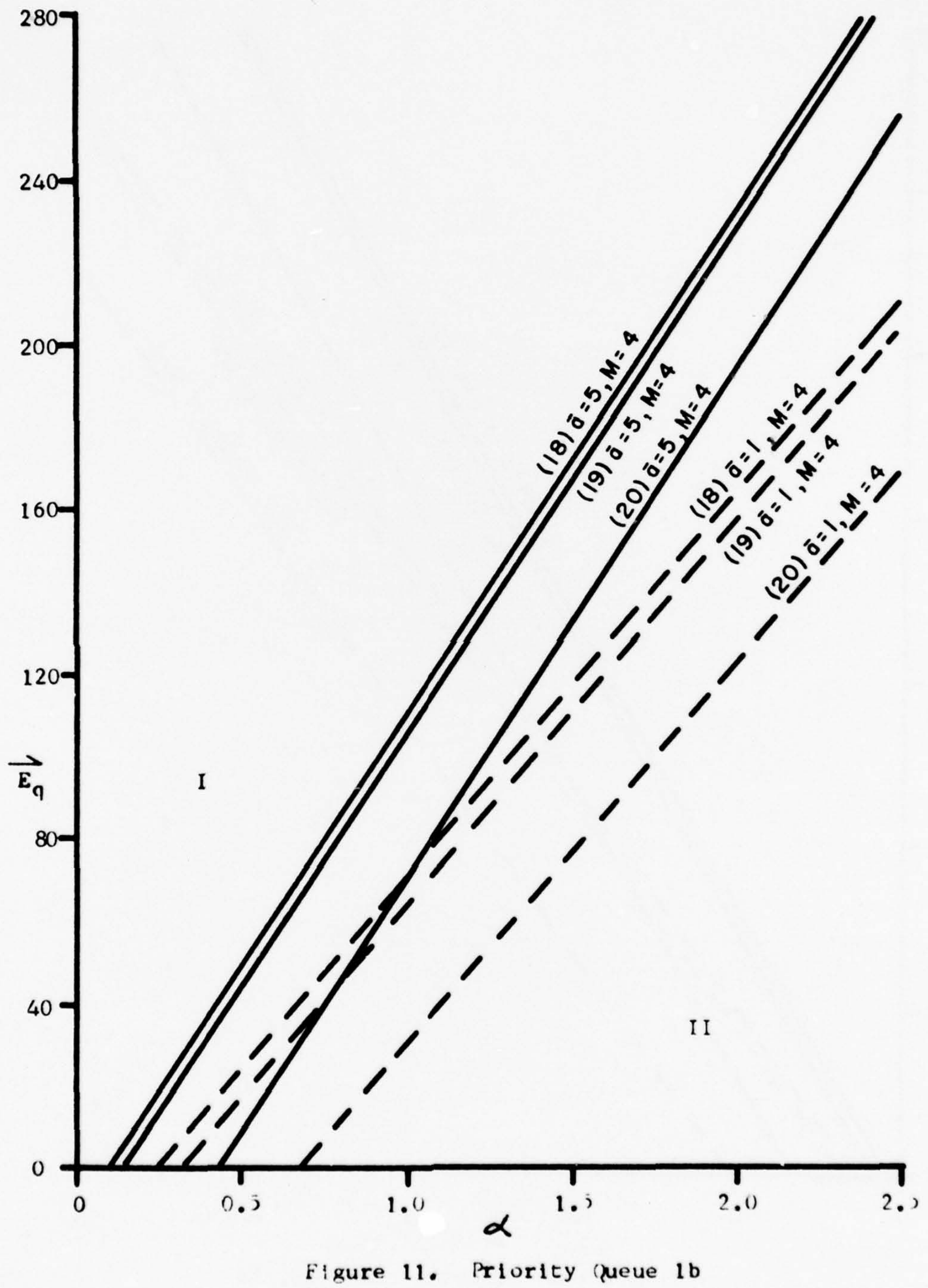


Figure 11. Priority Queue 1b

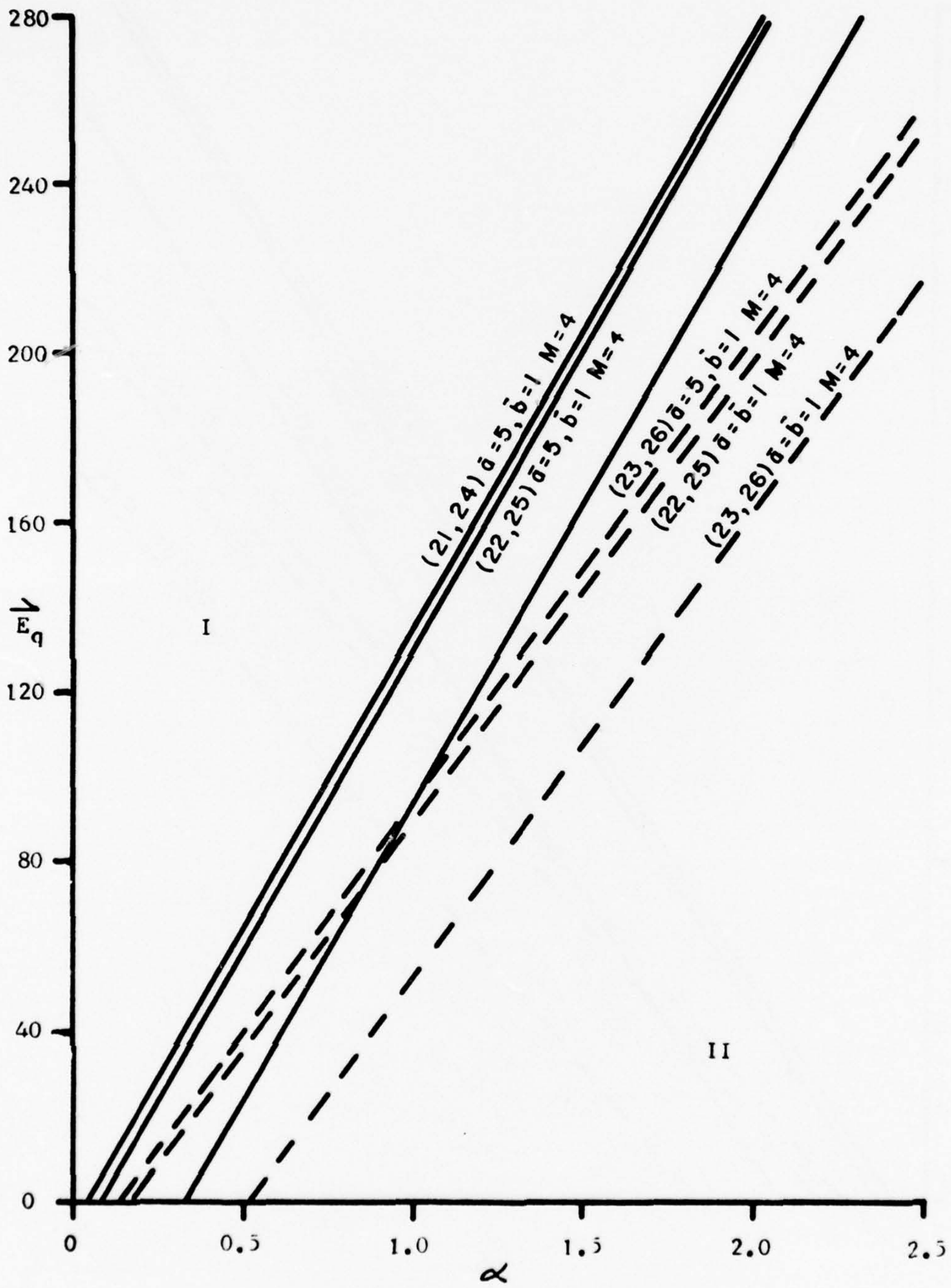


Figure 12. Priority Queue 2b, 3b

since a transfer to the AML must take place. Since the transfers are made by field, each individual field width must be accounted for separately. The series of equations follows the same three cases for the 'a' subcase in the sense that increasing field width is considered by adding respectively $\overrightarrow{Sk}$ and $\overrightarrow{Tk}$.

The general result in the 'b' subcase is that a multiple identical response is favorable to all the associative architectures because it is not necessary to repeat the composite node cycle for the M-1 additional responses as it would be in the RAM. Therefore, to arrive at the equations, the single node RAM time was increased by a factor of M, while only the allocate, insert and deallocate portions of the three associative architectures were increased by M. In the case of the AML, the transfer step is also included since it is part of the delete (search) step. Notice also in the graph that the vertical scale factor has been changed. As in the previous 'a' subcase, the AML curves represent lower overall values for $\overrightarrow{E_q}$ compared to the AM and AM/RAM cases (M = 4 in the graphs). The gamma and delta parameters can now be used to reduce the bit width requirements. A low value of M and b such as used here entirely favor the RAM architecture.

The last case, shown in Table 8, is the most interesting, since it indicates a sharp departure between the AM and the AML. In this category, the RAM, AM and AM/RAM total composite node cycle times were all increased by a factor of M. This means that there is no savings in time for the AM and AM/RAM architectures as there

Information Structure	Comparison	Equivalent Breakeven Bit Width Equations	Equation Number
Priority Queue 1c	RAM vs. AM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} = (2\bar{a}+21)\alpha + (\bar{a}-1)\beta - 7$	27
	RAM vs. AM/RAM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} = (2\bar{a}+21)\alpha + (\bar{a}-1)\beta - 9$	28
	RAM vs. AM/RAM/AML	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \left\lceil \frac{\vec{PK}}{\gamma} \right\rceil + \left\lceil \frac{\vec{PK}}{\delta} \right\rceil + \left\lceil \frac{\vec{AMA}}{\delta} \right\rceil = (2\bar{a}M+21M)\alpha + (\bar{a}M-4M-7)\beta - 13M$	29
Priority Queue 2c	RAM vs. AM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} + \vec{SK} = (2\bar{a}+3\bar{b}+22)\alpha + (\bar{a}+3\bar{b}-1)\beta - 7$	30
	RAM vs. AM/RAM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} + \vec{SK} = (2\bar{a}+3\bar{b}+22)\alpha + (\bar{a}+3\bar{b}-1)\beta - 9$	31
	RAM vs. AM/RAM/AML	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \left\lceil \frac{\vec{PK}}{\gamma} \right\rceil + \left\lceil \frac{\vec{PK}}{\delta} \right\rceil + \left\lceil \frac{\vec{SK}}{\delta} \right\rceil + \left\lceil \frac{\vec{AMA}}{\delta} \right\rceil = (2\bar{a}M+3\bar{b}M+22M)\alpha + (\bar{a}M+3\bar{b}M-3M-7)\beta - 13M$	32
Priority Queue 3c	RAM vs. AM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} + \vec{SK} + \vec{TK} = (2\bar{a}+3\bar{b}+22)\alpha + (\bar{a}+3\bar{b}-2)\beta - 7$	33
	RAM vs. AM/RAM	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \vec{PK} + \vec{SK} + \vec{TK} = (2\bar{a}+3\bar{b}+22)\alpha + (\bar{a}+3\bar{b}-2)\beta - 9$	34
	RAM vs. AM/RAM/AML	$\vec{E}_q = \left\lceil \frac{\vec{LN}}{\gamma} \right\rceil + \left\lceil \frac{\vec{PK}}{\gamma} \right\rceil + \left\lceil \frac{\vec{PK}}{\delta} \right\rceil + \left\lceil \frac{\vec{SK}}{\delta} \right\rceil + \left\lceil \frac{\vec{TK}}{\delta} \right\rceil + \left\lceil \frac{\vec{AMA}}{\delta} \right\rceil = (2\bar{a}M+3\bar{b}M+22M)\alpha + (\bar{a}M+3\bar{b}M-3M-7)\beta - 13M$	35

Table 8. $\vec{E}_q$ Equations for Priority Queues 1c, 2c, 3c

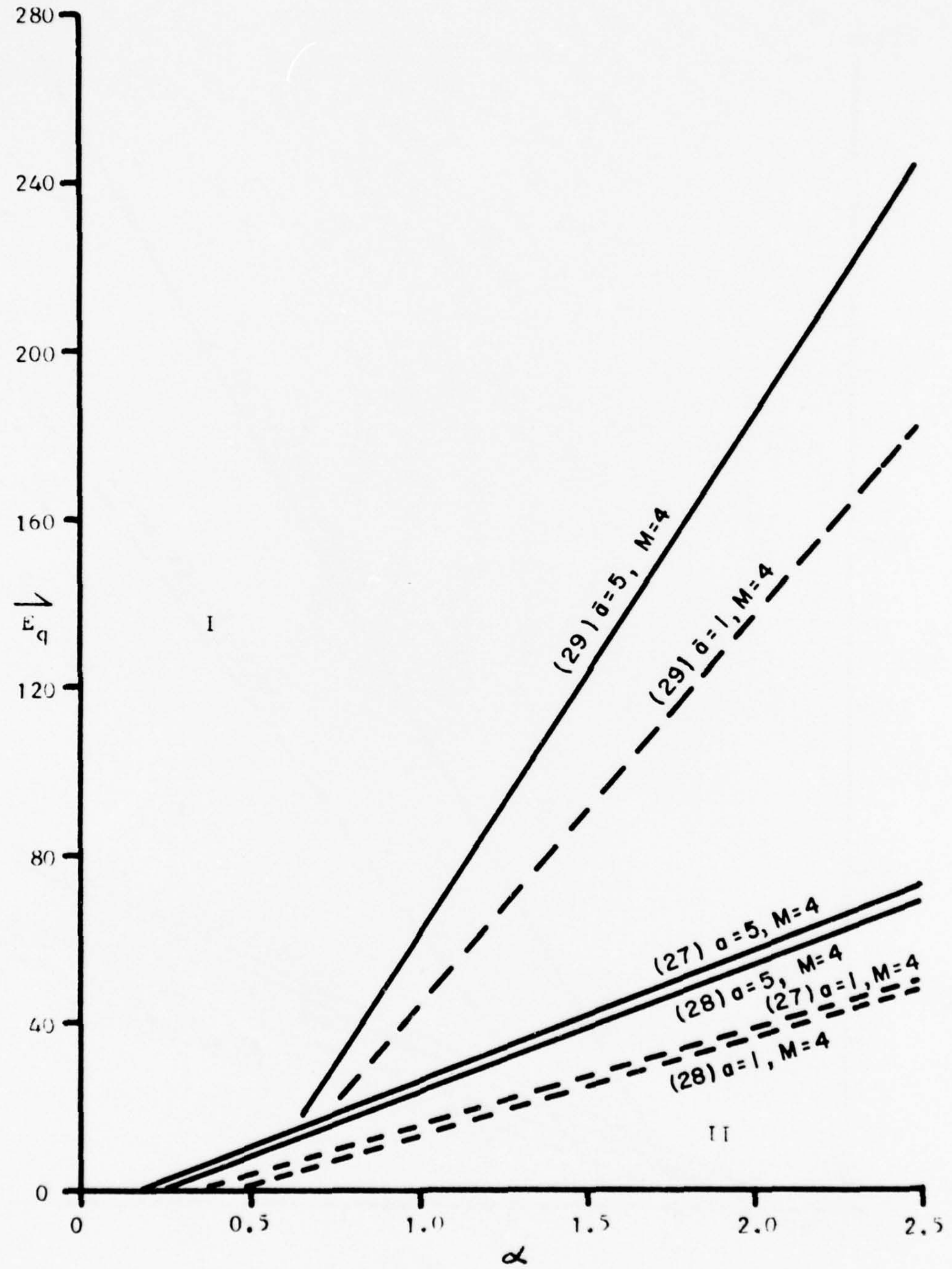


Figure 13. Priority Queue 1c

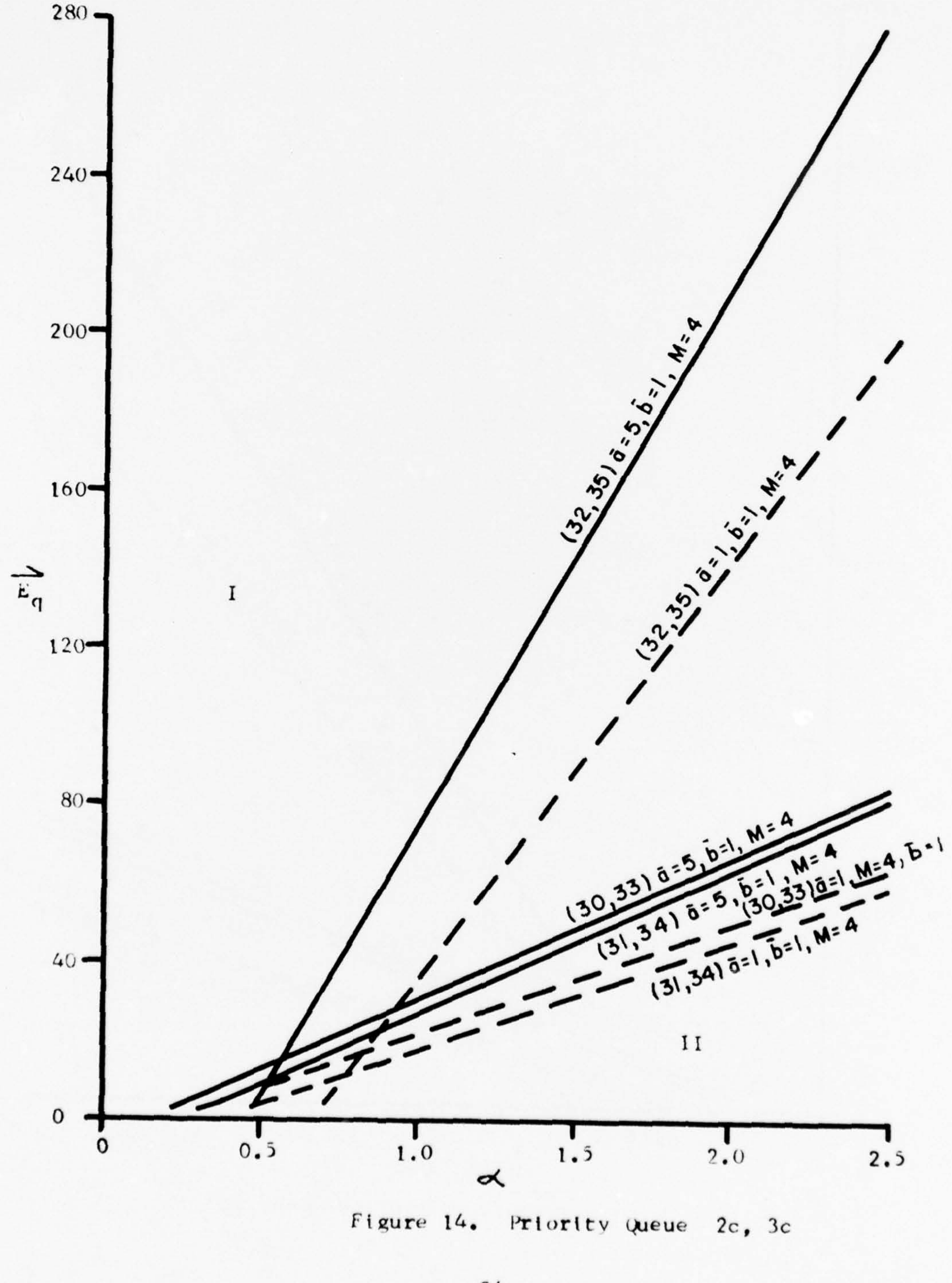


Figure 14. Priority Queue 2c, 3c

was in subcase 'b'. However, in the AML case the allocate, insert and deallocate times of the composite node cycle were increased by M as before, but now instead of increasing the delete time by M, it is only increased by an average of two EBW's. This is the average ordered retrieval time for Lewin's algorithm, since the total ordered retrieval time is 2M-1; hence retrieval time no longer increases linearly with M.

The poor results of the AM and AM/RAM architectures are plotted in Figures 13 and 14 (Equations 27, 28, 30, 31, 33 and 34). The mild slopes of these plots are indicative of the small number of bits available to meet all the field requirements listed in Table 8 as part of the equations. The problem of the small number of bits is compounded by the lack of parallelism available for primary, secondary and tertiary key searches.

In contrast to the results of the AM and AM/RAM, the AM/RAM/AML architecture results plotted in Figures 13 and 14 (Equations 29, 32 and 35) exhibit fairly steep slopes. This means that a relatively large number of bits is available to satisfy field requirements. Further, these field requirements, by virtue of the FIMV algorithm, exhibit full parallelism and hence require fewer bits than the AM and AM/RAM fields. As an example, consider Equation 29, plotted in Figure 13 for $M = 4$ and $a = \alpha = 1$. $\vec{E}_q$ in this case is forty bits. If gamma and delta were as little as two, this would allow twenty bits for each of the four fields that make up the total requirement for $\vec{E}_q$. In general this would be more than adequate. For Equations

32 and 35, plotted in Figure 14, the number of fields has increased by one and two respectively over Equation 29. Under the conditions of $M = 4$ and $a = b = \alpha = 1$, there are slightly less than forty bits to satisfy the total $\vec{E}_q$ requirements. This means that gamma and delta must be increased to three and perhaps four to maintain a performance equivalent to that of Equation 29. A more realistic set of parametric values for Equations 32 and 35 would be $M = 4$, $a = 5$, $b = 1$ for $\alpha = 1$. This represents the upper plot of Figure 14. Here seventy bits are available for distribution among five and six fields respectively. A value of two for gamma and delta would be quite adequate.

The major result then is that the FIMV algorithm coupled with the AML architecture does show a definite advantage over both the RAM and conventional AM and AM/RAM architecture. The b parameter was kept low to favor the random access architecture. In general, increasing b favors the associative architecture by increasing the sort-in time necessary for the RAM.

4.4. Priority Queues, Cases 4, 5, 6

This section covers three algorithms normally available as auxiliary algorithms in discrete simulation. These three are also referred to as random lists because they represent priority queues that are maintained in the RAM in an unordered fashion. This situation occurs in discrete simulation when it is necessary to search for and select a node in a priority queue that is not ranked on the search field. This situation also occurs in FTI time flow mechanisms where the list of future events is not ordered [44, 50].

The equations for the various information structures considered are listed in Table 9. The first random list considered (case 4) is one in which the list is searched to find the first node meeting some criterion. This may not be the only node meeting such a criterion, and the possibility of intentionally selecting more than one is covered in case 5. The RAM algorithm is based on the 'c' parameter, which is the expected list depth in nodes before a success. If the last node in the list were the only successful node, then c would equal $\overrightarrow{l_i}$. In the associative case the parameters are $\overrightarrow{LN}$ and $\overrightarrow{PK}$. Gamma affects both these parameters since they both can be used in a fully data parallel search. Figure 15 illustrates two families of curves. The first is for c = 1, the worst case comparison for the AM, and c = 5. At alpha equal to one, these two curves yield for the AM architecture seventeen and forty EBW's respectively. At gamma equal to two this becomes thirty-four and eight EBW's respectively, which even in the worst case would cover

Information Structure	Comparison	Equivalent Breakeven Bit Width Equations	Equation Number
Priority Queue	RAM vs. AM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \sqrt{\frac{\vec{PK}}{\gamma}} = (3c+20)\alpha + (2c-1)\beta - 6$	36
4	RAM vs. AM/RAM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \sqrt{\frac{\vec{PK}}{\gamma}} = (3c+20)\alpha + (2c-6)\beta - 8$	37
Priority Queue	RAM vs. AM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \sqrt{\frac{\vec{PK}}{\gamma}} = (3\vec{l}_i + d + 12)\alpha + (2\vec{l}_i + d - 1)\beta - 6$	38
5	RAM vs. AM/RAM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \sqrt{\frac{\vec{PK}}{\gamma}} = (3\vec{l}_i + d + 12)\alpha + (2l_i + d - 1)\beta - 8$	39
Priority Queue	RAM vs. AM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \vec{PK} = (3\vec{l}_i + 2e + 20)\alpha + (2\vec{l}_i + e - 3)\beta - 6$	40
6	RAM vs. AM/RAM	$\vec{E}_q = \sqrt{\frac{\vec{LN}}{\gamma}} + \vec{PK} = (3\vec{l}_i + 2e + 20)\alpha + (2\vec{l}_i + e - 2)\beta - 8$	41

Table 9. $\vec{E}_q$ Equations for Priority Queues 4, 5, 6

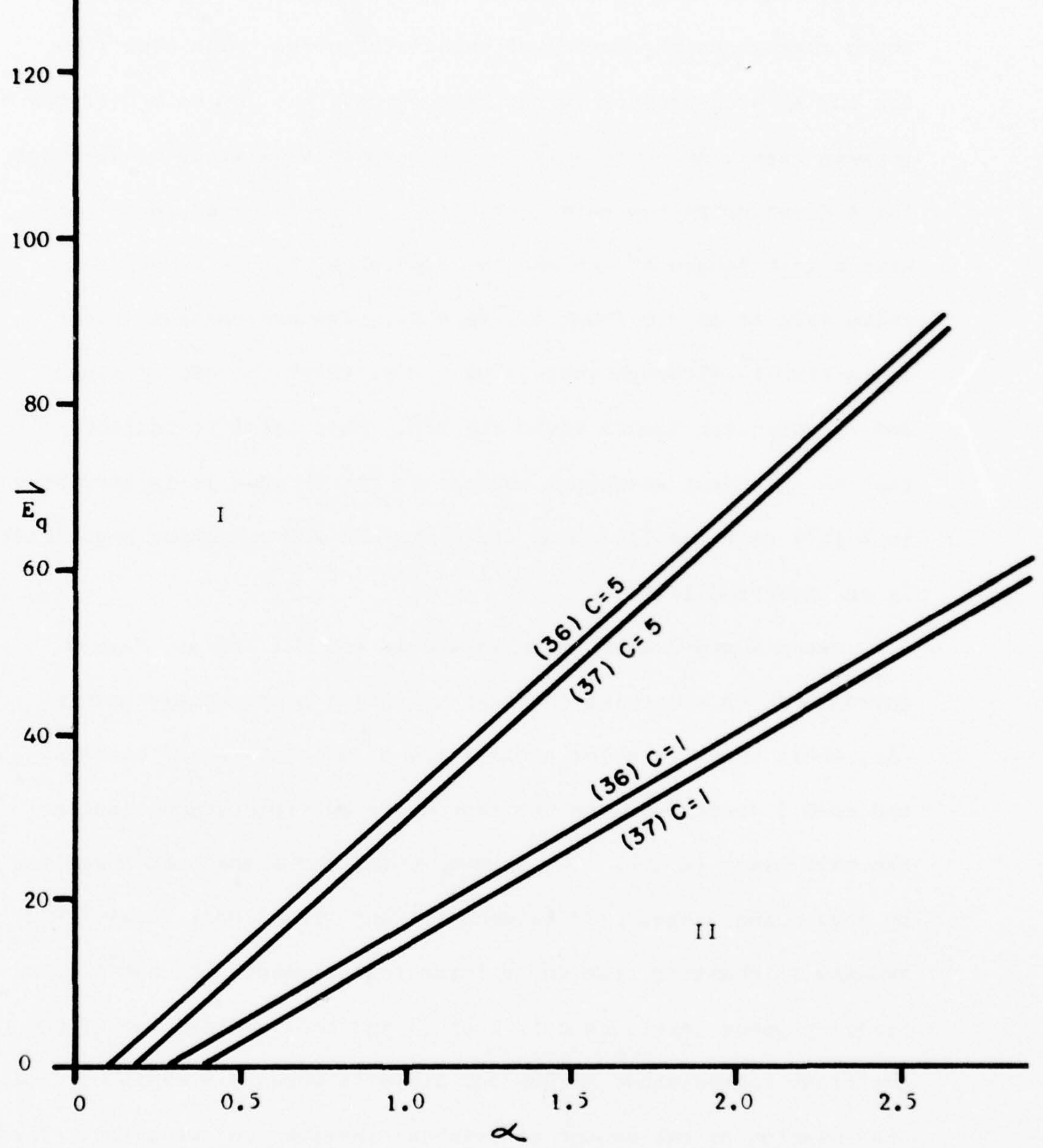


Figure 15. Priority Queue 4

a large dynamic range for the primary key.

In case 5 (Figure 16) the search is based on finding all nodes that meet some criterion. This means in the RAM architecture that the total list must be searched. This introduces the d parameter, which represents the number of successful nodes. The algorithm for the AM architecture is the same as case 4. The main difference between case 4 and case 5 is that the AM shows a greater advantage for a given operating point for five. In fact for a simple case with a list length of ten and two successes, $\vec{E}_q$ has a positive value even at $\alpha = 0$ (that is, $T_m = 0$). The predominant effect comes from $\vec{l}_i$ although increasing d does favor the AM. Again, d and $\vec{l}_i$ were kept low to favor the RAM. This seems to indicate that the greatest advantage accrues to the AM when it is searching in a full data parallel mode vis-a-vis the RAM searching sequentially an unordered list.

Cases 4 and 5 also serve as models for FTI TFM's. Case 4 corresponds to selecting the next potential state change within Δt, where there is a low probability of multiple state changes; and case 5 corresponds to the case where multiple state changes can take place in Δt. In essence, then, the changeover suggested by Morgan and Siegel [50] between FTI and VTI unconditional TFM's amounts to changing from the information structure of case 4 to a priority queue (e.g., case 1, 2 or 3) and back again. One of the questions raised about Morgan and Siegel's work by Wickham [71] was the question of the amount of overhead involved in switching. This

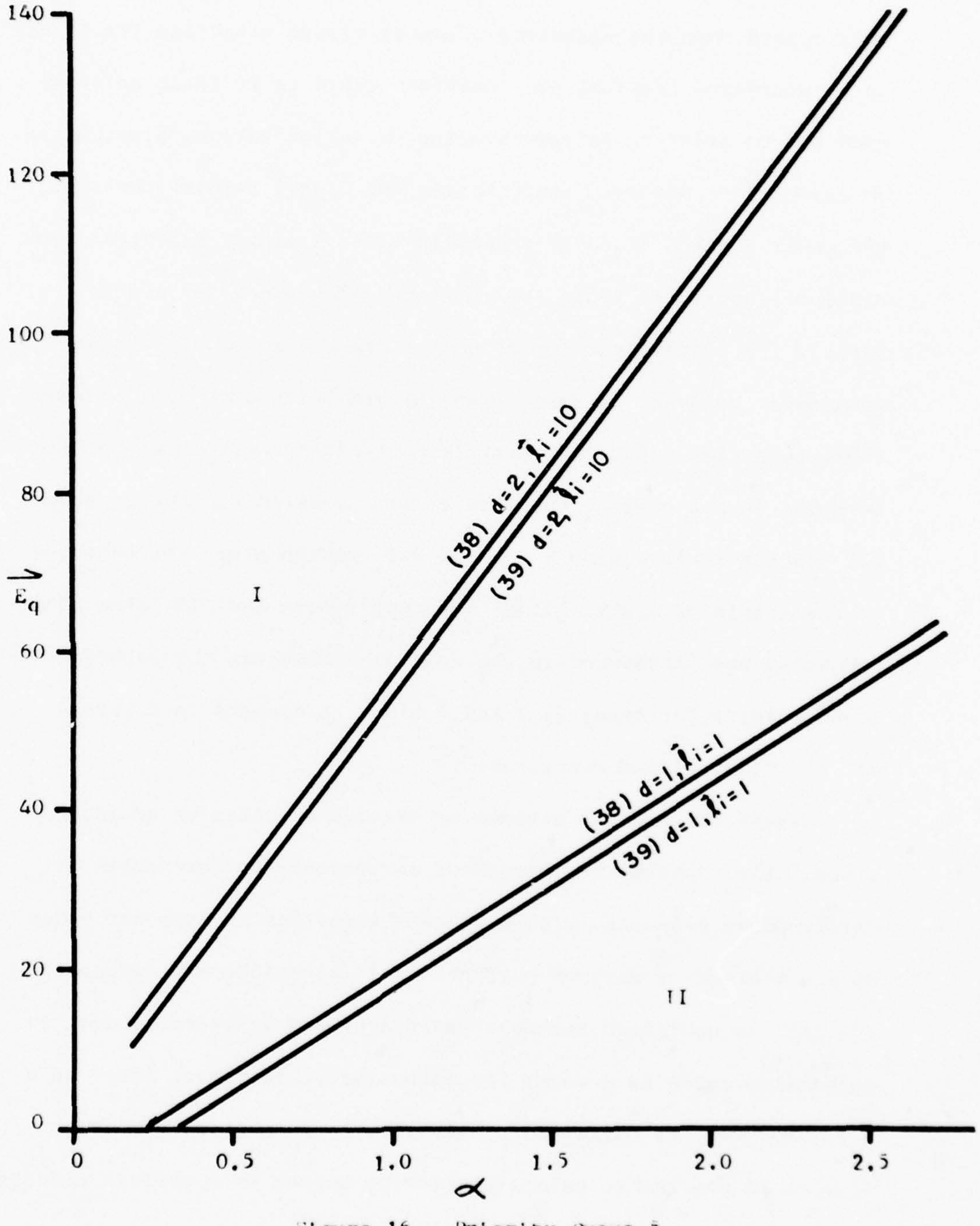


Figure 16. Priority Queue 5

overhead amounts to reordering the list when switching back from a random FTI list to a priority queue. It is interesting to note in this regard that the associative memory always maintains its lists in an unordered fashion, and therefore there is at least no overhead due to ordering in implementing Morgan and Siegel's method in an associative memory. Since Morgan and Siegel pointed out that the other portion of their scheme involves a simple numerical prediction process, it would seem that the AM would offer a good vehicle for reconsideration of this scheme. The major problem remaining, however, as Emshoff and Sisson point out [16], is that there is a time error and possibly a precedence error in the FTI method. In the case of the time error, Gafarian and Ancker point out that there is always a loss of information about the behavior of the simulated system [25]. For this reason the FIMV time flow mechanism was introduced in the section discussing the priority queue results for cases 1, 2 and 3 since it appears to alleviate all the difficulties mentioned.

Case 6 is based on minimum or maximum searches of unordered lists. The e parameter introduced corresponds to the number of interchanges necessary within the RAM algorithm to keep the value of the minimum or maximum current. This algorithm represents for the RAM the unordered method of maintaining a priority queue. It could also serve as a model for an unordered retrieval based on a FTI TFM. But, as suggested previously, it is generally more economical in the RAM to maintain priority queues by a sort-in process.

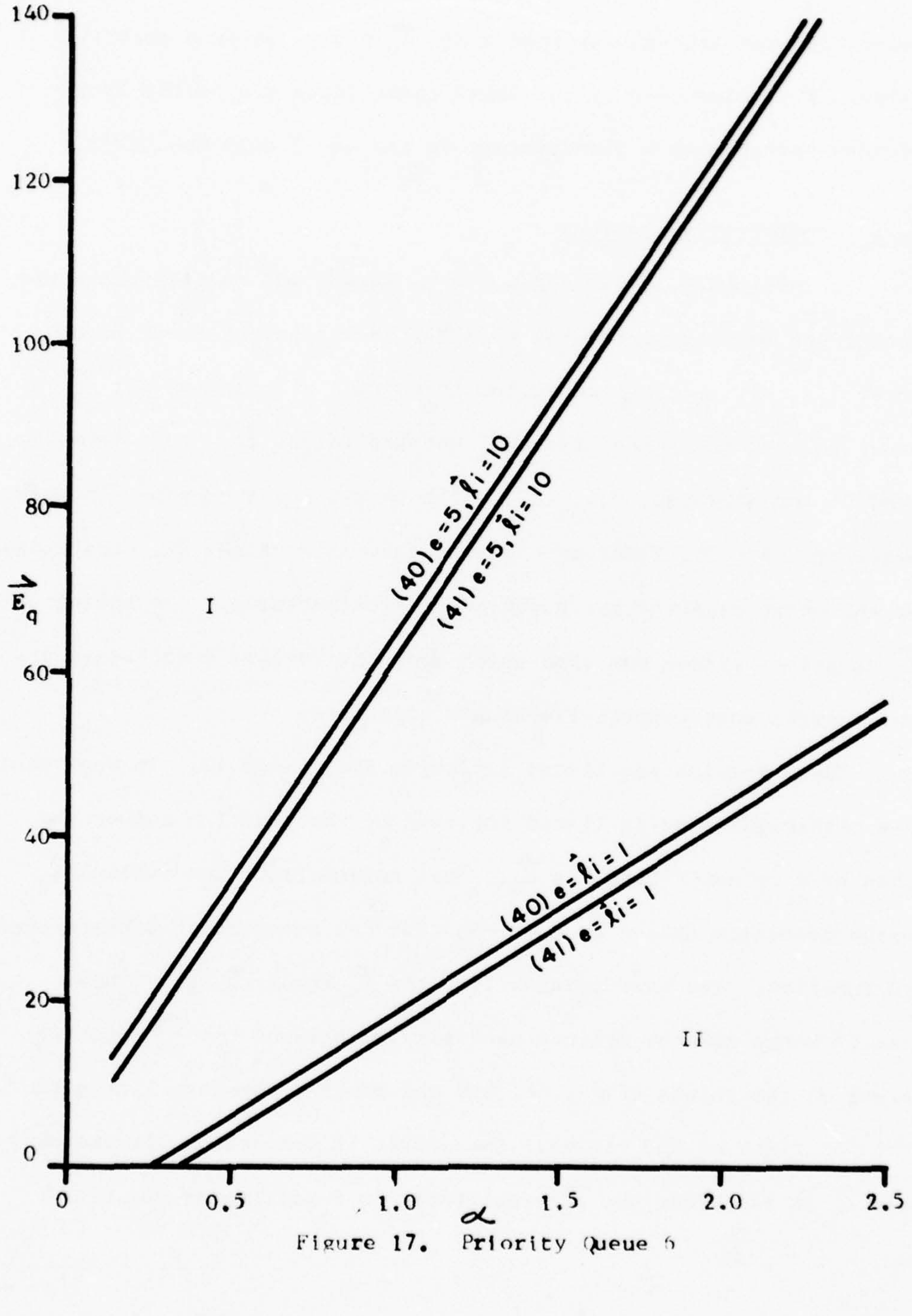


Figure 17. Priority Queue 6

Figure 17 illustrates some results for $\overline{E_q}$ based first on the worst case for the AM where $\overline{I_i}$=e=1 and a second case for $\overline{I_i}$=10 and e=5. Notice for the latter case that again $\overline{E_q}$ always shows a positive value. Note also that in the above three lists the AM/RAM architecture operates at a disadvantage to the AM of only two EBW's.

4.5. Parametric Variations

In sections 4.2, 4.3 and 4.4, the intent was to illustrate the method and results pertaining to a particular selection of parameters. In this section the intent is to take a subset of all $\overline{E_q}$ equations and study the effect of varying values for alpha (α=.75, α=1.0) and β (β=0, 0.5, 1.0) while permitting $\gamma = \xi = M_W$. The RAM parameters are all fixed at the most favorable values or, conversely, at the worst case for the associative architectures. The subset of $\overline{E_q}$ equations chosen was that which made the various associative architectures most competitive within each case.

These results are listed in Tables 10, 11 and 12. In each table the particular case is listed followed by the equation number and then by a column titled Min $\overline{E_q}$. This column lists the minimum $\overline{E_q}$ value necessary (based on $\gamma = \xi = M_W$) for the associative architecture to function. The next columns list the $\overline{E_q}$ requirements. These requirements are the minimum necessary to balance the $\overline{E_q}$ equations based on the values of α , β , and the other parameters listed to the far right of the tables. Therefore, in comparison, if the minimum $\overline{E_q}$ is less than the $\overline{E_q}$ requirement in a particular row, the

CASE	EQUATION	MIN $\overline{E_q}$ (1)	$\overline{E_q}$ REQUIREMENTS (2)			PARAMETERS	
			$\beta = 0.0$	$\beta = 0.5$	$\beta = 1.0$	α	OTHER
Queue - FIFO	3	2.0	3.0	1.0	-1.0	.75	
	AM	2.0	8.0	6.0	4.0	1.0	
Queue - FIFO	4	2.0	1.0	-1.0	-3.0	.75	
	AM/RAM	2.0	6.0	4.0	2.0	1.0	
Queue - LIFO	7	2.0	3.0	1.0	-1.0	.75	
	AM	2.0	8.0	6.0	4.0	1.0	
Queue - LIFO	8	2.0	1.0	-1.0	-3.0	.75	
	AM/RAM	2.0	6.0	4.0	2.0	1.0	

(1) Based on $\gamma = \delta = M_W$
(2) All values in EBW's

Table 10. Parameter Variations for Queues

CASE	EQUATION	MIN $\overline{E}_q$ (1)	$\overline{E}_q$ REQUIREMENTS (2)			PARAMETERS	
			$\beta = 0.0$	$\beta = 0.5$	$\beta = 1.0$	α	OTHER
Priority	11	2.0	8.0	5.0	2.0	.75	
Queue 1a			13.0	10.0	7.0	1.0	$a=1$
Priority	14	2.0	11.0	10.0	9.0	.75	
Queue 2a			17.0	16.0	15.0	1.0	$a=1, b=1$
Priority	17	2.0	11.0	10.0	9.0	.75	
Queue 3a			17.0	16.0	15.0	1.0	$a=1, b=1$
Priority	20	4.0	8M-4.0	8M-9.0	8M-14.0	.75	
Queue 1b			13M-4.0	13M-9.0	13M-14.0	1.0	$\bar{a}=1$
Priority	23	5.0	11M-4.0	13M-9.0	15M-14.0	.75	
Queue 2b			17M-4.0	19M-9.0	21M-14.0	1.0	$\bar{a}=1, \bar{b}=1$
Priority	26	6.0	11M-4.0	13M-9.0	15M-14.0	.75	
Queue 3b			17M-4.0	19M-9.0	21M-14.0	1.0	$\bar{a}=1, \bar{b}=1$
Priority	29	4.0	5M	3.5M-3.5	2M-7	.75	
Queue 1c			10M	8.5M-3.5	7M-7	1.0	$a=1$
Priority	32	5.0	8M	8.5M-3.5	9M-7	.75	
Queue 2c			14M	14.5M-3.5	15M-7	1.0	$\bar{a}=1, \bar{b}=1$
Priority	35	6.0	8M	8.5M-3.5	9M-7	.75	
Queue 3c			14M	14.5M-3.5	15M-7	1.0	$\bar{a}=1, \bar{b}=1$

(1) Based on $\gamma = \delta = M_w$
(2) All values in EBW's

Table 11. Parameter Variations for Priority Queues 1, 2, 3

CASE	EQUATION	MIN $\vec{E}_q$ (1)	$\vec{E}_q$ REQUIREMENTS (2)			PARAMETERS	
			$\beta = 0.0$	$\beta = 0.5$	$\beta = 1.0$	α	OTHER
Priority	36	2.0	12.0	12.5	13.0	.75	c=1
Queue 4	AM		17.0	17.5	18.0	1.0	
	37	2.0	10.0	10.5	11.0	.75	c=1
	AM/RAM		15.0	15.5	16.0	1.0	
Priority	38	2.0	6.0	7.0	8.0	.75	d=1, $\vec{I}_i$=1
Queue 5	AM		10.0	11.0	12.0	1.0	
	39	2.0	4.0	5.0	6.0	.75	d=1, $\vec{I}_i$=1
	AM/RAM		8.0	9.0	10.0	1.0	
Priority	40	1.0+$\vec{PK}$	12.0	12.5	13.0	.75	e=1, $\vec{I}_i$=1
Queue 6	AM		19.0	19.5	20.0	1.0	
	41	1.0+$\vec{PK}$	10.0	10.5	12.0	.75	e=1, $\vec{I}_i$=1
	AM/RAM		17.0	17.5	18.0	1.0	

(1) Based on $\gamma = \delta = M_w$
(2) All values in EBW's

Table 12. Parameter Variations for Priority Queues 4, 5, 6

associative architecture is superior. If the values are the same, the equation balances at the breakeven value; and if the requirement exceeds the minimum $\vec{E}_q$ value, then the RAM is superior.

Queue results are listed in Table 10. In ten out of the twelve cases for $\alpha=1$, the associative architectures, AM and AM/RAM, are superior to the RAM. In two cases, Equations 4 and 8 for $\beta=1.0$, they are equal to the RAM. If the associative compare time is decreased by 25% with respect to the RAM ($\alpha=.75$), the associative architectures are then superior in only two out of twelve cases ($\beta=0.0$). In the remaining ten cases the associative architectures are all inferior. This indicates that α plays a significant role, as suggested earlier.

Priority Queues 1, 2, 3 are covered in Table 11. All the results presented in Table 11 are based on the AM/RAM/AML architecture, and the a and b parameters all represent the worst case for this architecture. M is still left as a parameter.

For $\alpha=M=1.0$ in all but two cases, the AM/RAM/AML architecture is clearly superior to the RAM. In the other two cases, Equations 20 and 29 for $\beta=1.0$, $M=2$ is sufficient to make the AM/RAM/AML architecture superior. Even for a 25% reduction of relative compare time for the AM/RAM/AML architecture, all but one of the outcomes under subcases 1a, 2a and 3a are superior to the AM. The remaining case, Equation 11 at $\beta=1.0$ ties with the RAM.

With respect to subcases b and c under the 25% reduction, in all but two outcomes $M=2$ is sufficient to make the AM/RAM/AML superior to the RAM. In the remaining two outcomes, Equations 20 and 29 at $\beta = 1.0$, $M = 3$ is sufficient for the same result. Since $M = 4$, as referred to in Section 4.3, is considered to be a fair value for a discrete simulation, the AM/RAM/AML can easily sustaiin a 25% reduction in speed, and in selected cases for a higher value of M, even greater reductions. This illustrates another of the types of trade-offs available, which would be to go to a slower memory while increasing the processing width.

Table 12 lists the results for the remaining priority queues. The results for Priority Queues 4 and 5 indicate that the AM or AM/RAM architecture is superior to the RAM at both $\alpha = 1.0$ and $\alpha = .75$. Priority Queue 6 still relies on the $\gamma = 1$ minimum search, so PK is listed separately. For $\alpha = 1.0$, the AM and AM/RAM architecture should both be superior, assuming sixteen bits as an upper bound for $\overline{PK}$. The 25% reduction does generally make the RAM superior; however, in actual implementation the techniques of Priority Queue 1a should be substituted for those of Priority Queue 6.

The role of β as seen in Tables 10, 11 and 12 is one of biasing the results in favor of one of the architectures. Note, however, that the bias is not always in the same direction (e.g., Table 11). This phenomenon comes about as a result of the fact that sometimes the preponderance of overhead--non-memory instructions--lies with the associative architecture, and sometimes with the random access. Therefore β, as a measure of the degree of influence of this overhead, plays an important although not pre-emptive role in determining architecture behavior.

4.6. Summary

The main results are the various equations associated with the queue and priority queue information structures. These equations represent a source of considering not only the usefulness of the various architectures but of the tradeoffs associated with memory speed versus data parallelism. Since the equations do represent a rich source of information, the approach used in presenting the results was to illustrate the use of the equations by way of selected graphs and examples that pertain to discrete simulation instead of an exhaustive discourse. Specific conclusions and recommendations are presented in the next chapter.

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

5.1. General Conclusions

The general concern of this research is the utility of the associative memory for non-numeric processing in discrete simulation. The particular concern is priority queues, since they form the basis of the time flow mechanisms and are inherent in many simulation models. The research indicates that the associative memory can process priority queues more efficiently than the random access memory under the assumptions and constraints of the research. It further indicates that hybrid memories--such as the use of the random access memory in conjunction with the associative--is promising not only in terms of performance but also in terms of relative cost.

The research selected or created parametric hardware and software models that could be matched to each other to process a variety of priority queues. The hardware models consisted of a random access memory, an associative memory, a random access memory in conjunction with an associative memory, and a random access memory in conjunction with an associative memory with an added auxiliary memory implementing Lewin's minimum retrieval algorithm.

The key hardware parameters were the ratio of the random access memory speed to the associative memory speed (α); the ratio of the random access non-memory instruction time to the associative non-memory instruction time (β); and the degrée of parallelism in transferring data between various memories when they were used in a hybrid fasion, expressed as the number of bits simultaneously active (delta). The software parameters are related to individual priority queues of which the main ones are list length ($\overrightarrow{l_i}$) and the number of consecutive nodes retrieved in a given search (M).

The first queues studied were the simple LIFO and FIFO queues. They lend themselves only moderately well to associative memories. The major problem in fitting simple queues into associative memories is in obtaining a high degree of parallelism during the search for the next node. To obtain this parallelism the slow single bit at a time minimum associative search was converted into an "equal to" fully parallel associative search by changing the algorithm. Once this conversion was made and assuming the maximum value of gamma, the associative memory and the associative memory combined with the random memory proved to be good for alpha equal to or greater than one (Table 10). For alpha less than one the random access memory is definitely superior. The use of Lewin's minimum search algorithm is inappropriate although the algorithm controls processing time growth with list length. This is because the performance of the simple queues in the random access memory is independent of list length ($\overrightarrow{l_i}$).

The next group of priority queues deals with priority queues used as time flow mechanisms. Three versions of priority queues are considered in relation to situations normally found in discrete simulation. The associative and associative combined with random access memories proved to be of marginal utility compared to the random access memory alone. The combined associative, random and Lewin memory did show great utility when coupled with the fixed increment minimum value time flow mechanism algorithm. The results indicate that the associative memory combination can be 25% and in some cases 50% slower than the random access memory and still show an overall processing time advantage. It is also possible to reduce the degree of parallelism (gamma) from the maximum for alpha less than one without losing the associative advantage. This permits a certain degree of tradeoff in design parameters that offset overall cost. The FIMV TFM also seems to offer advantages discussed later for combining various types of discrete simulation TFM's into a unified model and simulation view.

The remaining priority queues studied were all concerned with various auxiliary operations with discrete simulation. In these cases the associative and associative combined with random access memories showed a marked advantage. This would permit good flexibility in design trade-offs. Lewin's algorithm was not investigated here because of the success of the conventional associative techniques.

In all priority queue cases studied the associative memory in one of the forms studied was equal to or better than the random access memory.

Conclusions About the Methodology

The methodology used in the research attempted to study the associative architecture in a complete way. Each individual algorithm was worked out and then combined with other algorithms to form composite node cycles. Total time measurements were then applied. In this way the fine structure of the computational process could be studied. For instance, the introduction of beta permitted a consideration of the contribution of non-memory instructions. In the cases of queues, beta quickly plays a decisive role in switching the superiority from the AM to the RAM at alpha less than one. The implication is that perhaps some more research is needed in preparing the data prior to search. In general, the use of the MIX computer approach quickly brought into focus the various areas that needed attention in the research.

Comparison to Other Work

As mentioned at the outset, there was no direct work in this area. Vaucher and Duval [69] did consider other RAM algorithmic methods for priority queues in discrete simulation. In their work they chose a particular implementation that did not use dynamic allocation but used a fixed list length. With some reservations, however, it is possible to make a limited comparison. For queues they found that the circular double linked linear list (CDLLL) was best for controlling time growth with list length for lengths of less than ten nodes. Subsequent techniques yielded better control on time growth, but always took more time with increasing length.

The queue techniques for the associative architecture used in this research were superior to the CDLLL. Further, processing time is independent of list length. For Vaucher and Duval's priority queues, priority queue case 1a seemed to be the closest match. This case assumes a single node selection. Further the list name search portion is dropped because Vaucher and Duval used single lists. This means that only the first part of the FIMV algorithm is used. The result is that the AM/RAM/AML is superior to the CDLLL used by Vaucher and Duval. Since all other algorithms used by them show a positive time growth with list length--and the technique used in this research does not--the associative architecture is definitely superior for maximum gamma and possibly for lesser values as well.

5.2. General Recommendations

There are several extensions to this research that are necessary to complete the assessment of associative architecture for discrete simulation. The first of these concerns the number of words in the memory. This research assumes that the memory was always big enough to contain the problem. This did not seem unreasonable in discrete simulation since information is being created and destroyed as opposed to only created and stored. Secondly it is assumed that the RAM also has sufficient memory. An investigation should be undertaken to consider memory overflow for both types of machines in relation to each other. Another area of concern is in the area of using non-uniform node size dynamic allocation schemes. Knuth [36] points out that t is is a very complex area for the RAM. A third area involves an expansion of Vaucher and Duval's work so that a better comparison can be made with more complex RAM algorithms. A fourth area would be a consideration of a few long words versus many short words. A preliminary look at this situation as part of the research indicated no clear-cut advantage either way, so for the sake of memory compatibility the short word was used. The exception to this was the memory for Lewin's algorithm because of the manner in which it works.

It is unlikely that research extended into these other areas based on this work can be done using the exact same methodology used here. Instead it is recommended that an emulator be established for the RAM based on MIX and the AM based perhaps on STARAN. Following this, formal simulations of those parts relating to the non-numeric processing should be set up to measure the relative performance of the machines with respect to the three areas above.

As a final recommendation on simulation and perhaps the most interesting, the merger of discrete and continuous simulation should be considered. This suggestion is based on the remark attributed to Gordon [26] that the FTI TFM would support either discrete or continuous simulation where VTI would support only discrete. Since the research demonstrated that an efficient TFM could be constructed by concatenating the FTI and VTI methods, it seems reasonable to suppose that the first part of the FIMV algorithm could support continuous simulation while the total algorithm would come into use for the discrete case. If this merger were possible, simulation could be used as a single efficient entity without regard to partitioning it into special categories of systems representation. One straightforward approach would be to apply the methodology of this research to GASP IV [61], a FORTRAN based simulation system which permits discrete and continuous phenomena to be modeled as a single system.

APPENDIX A. ASSOCIATIVE MEMORY

A simplified associative or content addressable memory (the terms are used interchangeably) is shown in Figure A-1. The method of using content addressability is as follows. Data encoded in binary form are stored in each memory word by a load operation such that the type of information is vertically aligned. As an example, consider billing information, where the first M bits of a word are reserved for name, the next N bits of the word are reserved for address, and the last K bits are reserved for the net amount owed, where each group of bits is called a field. To locate the account information of a particular person the comparand register is loaded with the name and the mask register disables all but the first M bits. An "equal to" search is conducted and the word which matches the proper name has its response bit set. The information can then be read out, updated, et cetera. A random access memory would need some indirect referencing scheme to locate the proper account and would generally be slower.

Now consider a slightly more complex situation where it is desired to locate all people who owe less than P dollars. Again the comparand is loaded with the search criterion--P dollars--and the mask register disables all but the last K bits. A "less than" search is then conducted on all words in parallel and the responders are set for all correct entries (words). The same task on a random access memory would be considerably more complicated than the first

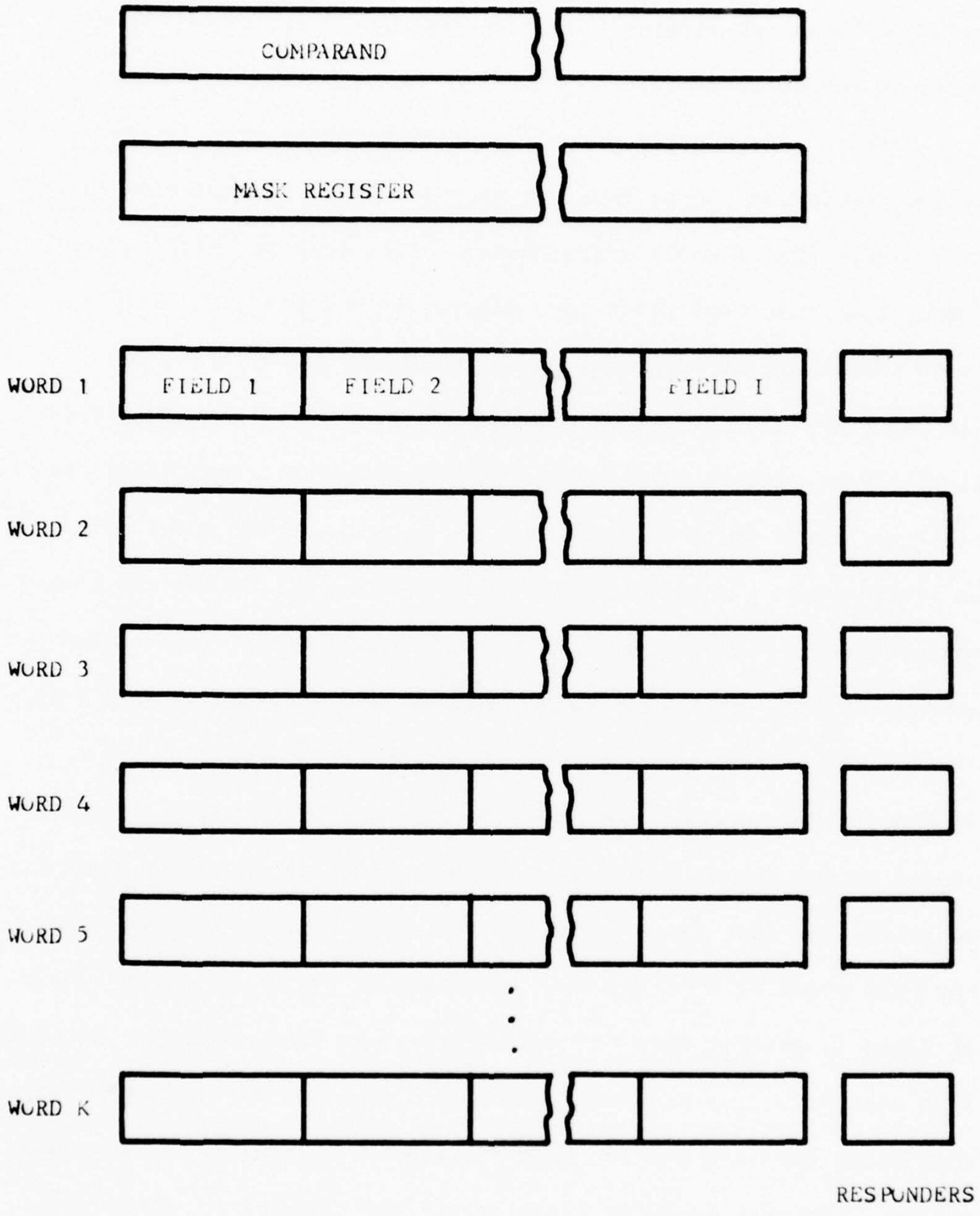


Figure A-1. Associative Memory

task. However, due to the content addressability of the associative system, this task is conceptually no more complicated than the first, and both tasks are simpler than even the first task conducted with a random access memory.

Consider now a third example dealing directly with discrete system simulation. A primary activity in discrete simulation is the time sequencing of model state changes. For each potential state change that must take place in a discrete simulation, there is a state change notice composed of the following parts, or fields (in the context of a content addressable memory). The first field is the event time or time of occurrence, while the second field is the event type, which determines which subpart of the model is to be exercised at the event time. The remaining fields are additional characteristics or attributes that provide additional information germane to that particular event, such as event priority, last time event occurred, system resources necessary for successful event completion, et cetera. Consider that all the event notices are stored in the content addressable memory and that the model must determine the next event under the following operating rule: find the next event of type "A" that possesses an attribute three (value of field 3) greater than "B" and an attribute five (value of field 5) less than "L". The process results in a complex search where fields one, three and five are set respectively to A, B and L in the comparand. The mask register disables all but fields one, three and five. The search would then typically proceed from left to right across the memory as follows: search on A to isolate words containing only events of type A. Then search on greater than B for those words surviving the first search on A and and the result with search on less than for L. Those words (event notices) surviving the search would have their responders set.

This simple example illustrates the usefulness of an associative memory for search operations, which comprise a large proportion of the actual execution of a discrete simulation. It should be pointed out that an associative processor can be considered to be an associative memory with arithmetic capability at each word.

What has been described above is what may be considered an explanation of a conventional associative array memory such as STARAN [63], although STARAN is actually an associative processor. There are two general differences between STARAN and the associative architecture used in this research. These differences will be discussed briefly below.

First, information is not stored horizontally by fields but vertically in nodes with one word allocated for each field. In terms of the last example, A might be stored in the first word, B in the second, and L in the third. The search would begin with A and all successes would be recorded in the response store. The search would then continue with a separate search instruction which would cause the response store result to be shifted down one so that the result of the search on A could be concatenated with the search on B and then shifted again for a separate search on L.

Second, the search process takes place with one or more bits at a time. For instance suppose each word were twenty-four bits long. Then the search could proceed from left to right (highest order bit to lowest order) γ bits at a time where γ could be one, two, ..., twenty-four. Therefore if γ were two, it would take twelve memory interrogations to complete the search for A or B or L. That means thirty-six memory interrogations for all three. The search takes place in a regular parallel fashion which means that the same bits in each word are compared at the same time with the comparand. Gamma can vary from one up to the word width for all comparand searches, that is, searches where each word is compared with the comparand independently of the others, such as a search for greater than or less than. In cases such as finding the maximum or minimum, other procedures must be used; and they are discussed in the body of the dissertation.

APPENDIX B. ALGORITHMS

B.1. General

This appendix discusses the various algorithms and their associated timings which were used to develop the results of Chapter IV. Additional background material germane to the use of the algorithms, such as additional conditions, is also discussed along with the algorithms. Each algorithm is discussed individually along with its flow chart. The MIX documentation [37] should be reviewed before studying the algorithms. Prior to that there are four summary tables that are discussed below.

Table B-1 tabulates the total composite node cycle time for each architecture studied for queues. The four cases discussed in the table are the same four cases by number discussed in Chapters III and IV. On the right hand side of the table are listed the algorithms that make up the composite node cycle for that architecture. Each algorithm so referenced can be looked up in Table B-4 for its individual timing and the figure that giver that algorithm's flow chart. The abbreviations for the algorithms are A for allocate, I for insert, S for search, D for delete, and DA for deallocate; primes indicate associative algorithms.

Tables B-2 and B-3 contain the same type of information as Table B-1. Table B-2 covers priority queue cases 1, 2 and 3 and also lists the subcases discussed in Chapter III, which are again covered in more detail as part of the discussions of algorithms

Information Structure	Architecture	Total Composite Node Cycle Time (TT_i)	Composite Node Cycle
Queue (FIFO) 1	RAM	$20T_M+6T_A$	$A_1-I_1-D_1-DA_1$
	AM	$(8+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK})T'_M+8T'_A$	$A'_1-I'_2-D'_3-DA'_1$
	AM/RAM	$(10+\overline{\frac{\overrightarrow{LN}}{\gamma}}+\overrightarrow{PK}(T'_M+8T'_A$	$A'_2-I'_2-D'_4-DA'_1$
Queue (FIFO) 2	RAM	$20T_M+6T_A$	$A_1-I_1-D_1-DA_1$
	AM	$(12+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{C'_j}}{\gamma}\right.)T'_M+10T'_A$	$A'_1-I'_3-D'_1-DA'_1$
	AM/RAM	$(14+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{C'_j}}{\gamma}\right.)T'_M+10T'_A$	$A'_2-I'_3-D'_2-DA'_1$
Queue (LIFO) 3	RAM	$20T_M+6T_A$	$A_1-I_1-D_1-DA_1$
	AM	$(8+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK})T_M+8T'_A$	$A'_1-I'_2-D'_3-DA'_1$
	AM/RAM	$(10+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK})T'_M+8T'_A$	$A'_2-I'_2-D'_4-DA'_1$
Queue (LIFO) 4	RAM	$20T_M+6T_A$	$A_1-I_1-D_1-DA_1$
	AM	$(12+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{C'_j}}{\gamma}\right.)T'_M+10T'_A$	$A'_1-I'_3-D'_1-DA'_1$
	AM/RAM	$(14+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{C'_j}}{\gamma}\right.)T'_M+10T'_A$	$A'_2-I'_3-D'_2-DA'_1$

Table B-1. Composite Node Cycle Times (Queues)

Information Structure	Architecture	Total Composite Node Cycle Time (TT_i)	Composite Node Cycle
1a	RAM	$(2a+21)T_M+(a+7)T_A$	$A_1-I_2-D_1-DA_1$
	AM	$(7+\lceil \overrightarrow{LN}/\gamma+\overrightarrow{PK})T'_M+8T'_A$	$A'_1-I'_1-D'_3-DA'_1$
	AM/RAM	$(9+\lceil \overrightarrow{LN}/\gamma+\overrightarrow{PK})T'_M+8T'_A$	$A'_2-I'_1-D'_4-DA'_1$
	AM/RAM/AML	$(10+\lceil \overrightarrow{LN}/\gamma+\lceil \overrightarrow{PK}/\gamma)T'_M+14T'_A$	$A'_3-I'_1-D'_9-DA'_2$
2a	RAM	$(2a+3b+22)T_M+(a+3b+8)T_A$	$A_1-I_3-D_1-DA_1$
	AM	$(7+\lceil \overrightarrow{LN}/\gamma+\overrightarrow{PK}+\overrightarrow{SK})T'_M+9T'_A$	$A'_1-I'_1-D'_5-DA'_1$
	AM/RAM	$(9+\lceil \overrightarrow{LN}/\gamma+\overrightarrow{PK}+\overrightarrow{SK})T'_M+9T'_A$	$A'_2-I'_1-D'_6-DA'_1$
	AM/RAM/AML	$(10+\lceil \overrightarrow{LN}/\gamma+\lceil \overrightarrow{PK}/\gamma)T'_M+14T'_A$	$A'_3-I'_1-D'_{10}-DA'_2$
3a	RAM	$(2a+3b+22)T_M+(a+3b+8)T_A$	$A_1-I_3-D_1-DA_1$
	AM	$(7+\lceil \overrightarrow{LN}/\gamma+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+10T'_A$	$A'_1-I'_1-D'_7-DA'_1$
	AM/RAM	$(9+\lceil \overleftarrow{LN}/\gamma+\overleftarrow{PK}+\overleftarrow{SK}+\overleftarrow{TK})T'_M+10T'_A$	$A'_2-I'_1-D'_8-DA'_1$
	AM/RAM/AML	$(10+\lceil \overleftarrow{LN}/\gamma+\lceil \overleftarrow{PK}/\gamma)T'_M+14T'_A$	$A'_3-I'_1-D'_{11}-DA'_2$

Table B-2. Composite Node Cycle Times (Priority Queue 1, 2, 3)

Information Structure	Architecture	Total Composite Node Cycle Time (TT_i)	Composite Node Cycle
1b	RAM	$M(2\bar{a}+21)T_M+M(\bar{a}+7)T_A$	1a
	AM	$(6M+1+\lceil \overrightarrow{LN}/\gamma \rceil+\overrightarrow{PK})T'_M+(5M+3)T'_A$	1a
	AM/RAM	$(7M+2+\lceil \overrightarrow{LN}/\gamma \rceil+\overrightarrow{PK})T'_M+(5M+3)T'_A$	1a
	AM/RAM/AML	$(10M+4+\lceil \overrightarrow{LN}/\gamma \rceil+\lceil \overrightarrow{PK}/\gamma \rceil+\lceil \overrightarrow{PK}/\varsigma \rceil+\lceil \overrightarrow{AMA}/\varsigma \rceil)T'_M+(8M+10)T'_A$	1a
2b	RAM	$M(2\bar{a}+3\bar{b}+22)T_M+M(\bar{a}+3\bar{b}+8)T_A$	2a
	AM	$(6M+1+\lceil \overrightarrow{LN}/\gamma \rceil+\overrightarrow{PK}+\overrightarrow{SK})T'_M+(5M+4)T'_A$	2a
	AM/RAM	$(7M+2+\lceil \overrightarrow{LN}/\gamma \rceil+\overrightarrow{PK}+\overrightarrow{SK})T'_M+(5M+4)T'_A$	2a
	AM/RAM/AML	$(10M+4+\lceil \overrightarrow{LN}/\gamma \rceil+\lceil \overrightarrow{PK}/\gamma \rceil+\lceil \overrightarrow{PK}/\varsigma \rceil+\lceil \overrightarrow{SK}/\varsigma \rceil+\lceil \overrightarrow{AMA}/\varsigma \rceil)T'_M+(8M+10)T'_A$	2a
3b	RAM	$M(2\bar{a}+3\bar{b}+22)T_M+M(\bar{a}+3\bar{b}+8)T_A$	3a
	AM	$(6M+1+\lceil \overrightarrow{LN}/\gamma \rceil+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+(5M+5)T'_A$	3a
	AM/RAM	$(7M+2+\lceil \overrightarrow{LN}/\gamma \rceil+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+(5M+5)T'_A$	3a
	AM/RAM/AML	$(10M+4+\lceil \overrightarrow{LN}/\gamma \rceil+\lceil \overrightarrow{PK}/\gamma \rceil+\lceil \overrightarrow{PK}/\varsigma \rceil+\lceil \overrightarrow{SK}/\varsigma \rceil+\lceil \overrightarrow{TK}/\varsigma \rceil+\lceil \overrightarrow{AMA}/\varsigma \rceil)T'_M+(8M+10)T'_A$	3a

Table B-2. Composite Node Cycle Times (Priority Queue 1, 2, 3) (Continued)

Information Structure	Architecture	Total Composite Node Cycle Time (TT_i)	Composite Node Cycle
1c	RAM	$M[(2\bar{a}+21)T_M+(\bar{a}+7)T_A]$	1a
	AM	$M[(7+\sqrt{\overrightarrow{LN}/\gamma}+\overrightarrow{PK})T'_M+8T'_A]$	1a
	AM/RAM	$M[(9+\sqrt{\overrightarrow{LN}/\gamma}+\overrightarrow{PK})T'_M+8T'_A]$	1a
	AM/RAM/AML	$(13M+\sqrt{\overrightarrow{LN}/\gamma}+\sqrt{\overrightarrow{PK}/\gamma}+\sqrt{\overrightarrow{PK}/\zeta}+\sqrt{\overrightarrow{AMA}/\zeta})T'_M+(11M+7)T'_A$	1a
2c	RAM	$M[(2\bar{a}+3\bar{b}+22)T_M+(\bar{a}+3\bar{b}+8)T_A]$	2a
	AM	$M[(7+\sqrt{\overrightarrow{LN}/\gamma}+\overrightarrow{PK}+\overrightarrow{SK})T'_M+9T'_A]$	2a
	AM/RAM	$M[(9+\sqrt{\overrightarrow{LN}/\gamma}+\overrightarrow{PK}+\overrightarrow{SK})T'_M+9T'_A]$	2a
	AM/RAM/AML	$(13M+\sqrt{\overrightarrow{LN}/\gamma}+\sqrt{\overrightarrow{PK}/\gamma}+\sqrt{\overrightarrow{PK}/\zeta}+\sqrt{\overrightarrow{SK}/\zeta}+\sqrt{\overrightarrow{AMA}/\zeta})T'_M+(11M+7)T'_A$	2a
3c	RAM	$M[(2\bar{a}+3\bar{b}+22)T_M+(\bar{a}+3\bar{b}+8)T_A]$	3a
	AM	$M[(7+\sqrt{\overrightarrow{LN}/\gamma}+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+10T'_A]$	3a
	AM/RAM	$M[(9+\sqrt{\overrightarrow{LN}/\gamma}+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+10T'_A]$	3a
	AM/RAM/AML	$(13M+\sqrt{\overrightarrow{LN}/\gamma}+\sqrt{\overrightarrow{PK}/\gamma}+\sqrt{\overrightarrow{PK}/\zeta}+\sqrt{\overrightarrow{SK}/\zeta}+\sqrt{\overrightarrow{TK}/\zeta}+\sqrt{\overrightarrow{AMA}/\zeta}(T'_M+(11M+7)T'_A$	3a

Table B-2. Composite Node Cycle Times (Priority Queue 1, 2, 3) (Completed)

Information Structure	Architecture	Total Composite Node Cycle Time (TT_i)	Composite Node Cycle
4	RAM	$(3c+20)T_M+(2c+7)T_A$	$A_1-I_1-S_1-D_2-DA_1$
	AM	$(6+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)T'_M+8T'_A$	$A'_1-I'_1-S'_1-DA'_1$ *
	AM/RAM	$(8+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)T'_M+8T'_A$	$A'_2-I'_1-S'_3-DA'_1$ *
5	RAM	$(3l_i+d+21)T_M+(2l_i+d+7)T_A$	$A_1-I_1-S_2-D_2-DA_1$
	AM	$(6+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)T'_M+8T'_A$	$A'_1-I'_1-S'_1-DA'_1$ *
	AM/RAM	$(8+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)T'_M+8T'_A$	$A'_2-I'_1-S'_3-DA'_1$ *
6	RAM	$(3l_i+2e+20)T_M+(2l_i+e+5)T_A$	$A_1-I_1-S_3-D_2-DA_1$
	AM	$(6+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK})T'_M+(7)T'_A$	$A'_1-I'_1-S'_2-DA'_1$ *
	AM/RAM	$(8+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK})T'_M+(7)T'_A$	$A'_2-I'_1-S'_4-DA'_1$ *

*Indicates delete algorithm not used since the node address is known by virtue of the search algorithm.

Table B-3. Composite Node Cycle Times (Priority Queue 4, 5, 6)

D'_9, D'_{10}, and D'_{11}. Where the same composite node cycles apply to a subcase b or c, as they do for subcase a, the reference subcase is entered in the composite node cycle column as opposed to relisting the cycle members.

Table B-3 lists the balance of the cases for priority queues. This table contains the same type of information as the previous two tables.

Additional algorithmic information is listed in Table B-4. The first part of Table B-4 lists all the RAM algorithms (unprimed) with the balance of the table used to list associative algorithms (primed). The appropriate figure for each algorithm is given and the subcases discussed above are cross-referenced in the remarks column.

The flow charts use standard MIX notation with the instruction time included in the lower part of the instruction symbol.

B.2. MIX-RAM Algorithms

This section contains flow diagrams and descriptions for algorithms A_1, DA_1, I_1, I_2, I_3, D_1, D_2, S_1, S_2, and S_3. These algorithms are adapted from Knuth, Volume I, Sections 2.2.3 and 2.2.5 [36]. Although these algorithms assume that LLINK and RLINK are stored in word zero (first word) of the node, no difference in timing occurs if each link occupies a separate word in core.

Algorithm A_1

This algorithm, shown in Figure B-1, is designed to allocate a node of a uniform number of words. The algorithm is independent of the eventual use of the node. The algorithm assumes that there is

Algorithm	Timing	Remarks	Figure
A_1		Prior Linkage of AVAIL List	B-1
	$4T_M+2T_A$	Normal	
	$4T_M+2T_A$	Subcase a	
	$M(4T_M+2T_A)$	Subcase b,c	
DA_1	$3T_M+T_A$	Normal	B-2
	$3T_M+T_A$	Subcase a	
	$M(3T_M+T_A)$	Subcase b,c	
I_1	$6T_M+T_A$		B-3
I_2	$(7+2a)T_M+(2+a)T_A$	Normal	B-4
	$(7+2a)T_M+(2+a)T_A$	Subcase 1a	
	$M\left[(7+2a)T_M+(2+a)T_A\right]$	Subcase 1b,1c	
I_3	$(8+3b+2a)T_M+(3+3b+a)T_A$	Normal	
	$(8+3b+2a)T_M+(3+3b+a)T_A$	Subcase 2a,3a	
	$M\left[(8+3b+2a)T_M+(3+3b+a)T_A\right]$	Subcase 2b,3b,2c,3c	
D_1	$7T_M+2T_A$	Normal	B-5
	$7T_M+2T_A$	Subcase a	
	$M(7T_M+2T_A)$	Subcase b,c	
D_2	$5T_M+T_A$	Normal	B-6

Table B-4. Algorithm Timings

Algorithm	Timing	Remarks	Figure
S_1	$(2+3c)T_M+(2+2c)T_A$	Normal	B-7
S_2	$(3+d+3\vec{I}_i)T_M+(2+d+2\vec{I}_i)T_A$	Normal	B-7
S_3	$(2+2e+3\vec{I}_i)T_M+(e+2\vec{I}_i)T_A$	Normal	B-8
A'_1	$3T'_M+3T'_A$	Normal	B-9
	$3T'_M+3T'_A$	Subcase a	
	$M(3T'_M+3T'_A)$	Subcase b,c	
A'_2	$4T'_M+3T'_A$	Normal	B-9
	$4T'_M+3T'_A$	Subcase a	
	$M(4T'_M+3T'_A)$	Subcase b,c	
A'_3	$5T'_M+4T'_A$	Normal	B-10
	$5T'_M+4T'_A$	Subcase a	
	$M(5T'_M+4T'_A)$	Subcase b, c	
I'_1	$2T'_M+2T'_A$	Normal	B-11
	$2T'_M+2T'_A$	Subcase a	
	$M(2T'_M+2T'_A)$	Subcase b, c	
I'_2	$3T'_M+2T'_A$	Normal	B-12
I'_3	$5T'_M+3T'_A$	Normal	B-13

Table B-4. Algorithm Timings (page 2)

Algorithm	Timing	Remarks	Figure
D'_1	$(3+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\left\lceil\frac{\overrightarrow{C'_j}}{\gamma}\right\rceil)\Gamma'_M+4\Gamma'_A$	Normal	B-14
D'_2	$(4+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\left\lceil\frac{\overrightarrow{C'_j}}{\gamma}\right\rceil)\Gamma'_M+4\Gamma'_A$	Normal	B-14
D'_3	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK})\Gamma'_M+3\Gamma'_A$	Normal	B-15
	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK})\Gamma'_M+3\Gamma'_A$	Subcase 1a, 1b	
	$M\left[(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK})\Gamma'_M+3\Gamma'_A\right]$	Subcase 1c	
D'_4	$(2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK})\Gamma'_M+3\Gamma'_A$	Normal	B-15
	$(2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK})\Gamma'_M+3\Gamma'_A$	Subcase 1a, 1b	
	$M\left[(2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK})\Gamma'_M+3\Gamma'_A\right]$	Subcase 1c	
D'_5	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK}+\overrightarrow{SK})\Gamma'_M+4\Gamma'_A$	Normal	B-16
	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK}+\overrightarrow{SK})\Gamma'_M+4\Gamma'_A$	Subcase 2a, 2b	
	$M\left[(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK}+\overrightarrow{SK})\Gamma'_M+4\Gamma'_A\right]$	Subcase 2c	
D'_6	$(2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK}+\overrightarrow{SK})\Gamma'_M+4\Gamma'_A$	Normal	B-16
	$(2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK}+\overrightarrow{SK})\Gamma'_M+4\Gamma'_A$	Subcase 2a, 2b	
	$M\left[(2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right\rceil+\overrightarrow{PK}+\overrightarrow{SK})\Gamma'_M+4\Gamma'_A)\right]$	Subcase 2b	

Table B-4. Algorithm Timings (page 3)

Algorithm	Timing	Remarks	Figure
D'_7	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+5T'_A$	Normal	B-17
	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+5T'_A$	Subcase 3a,3b	
	$M\left[(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+5T'_A\right]$	Subcase 3c	
D'_8	$(2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+5T'_A$	Normal	B-17
	$(2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+5T'_A$	Subcase 3a,3b	
	$M\left[2+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK}+\overrightarrow{SK}+\overrightarrow{TK})T'_M+5T'_A\right]$	Subcase 3c	
D'_9	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)T'_M+6T'_A$	Normal	B-18
	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)T'_M+6T'_A$	Subcase 1a	
	$(4+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{AMA}}{\varsigma}\right.)T'_M+10T'_A$	Subcase 1b	
	$(4M+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{AMA}}{\varsigma}\right.)T'_M$ $+(4M+7)T'_A$	Subcase 1c	
D'_{10}	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)T'_M+6T'_A$	Normal	B-18
	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)T'_M+6T'_A$	Subcase 2a	
	$(4+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{SK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{AMA}}{\varsigma}\right.)T'_M$ $+10T'_A$	Subcase 2b	
	$(4M+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{SK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{AMA}}{\varsigma}\right.)T'_M$ $+(4M+7)T'_A$	Subcase 2c	

Table B-4. Algorithm Timings (page 4)

Algorithm	Timing	Remarks	Figure
D'_{11}	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)\Gamma'_M+6\Gamma'_A$	Normal	B-18
	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)\Gamma'_M+6\Gamma'_A$	Subcase 3a	
	$(4+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{SK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{TK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{AMA}}{\varsigma}\right.)\Gamma'_M$ $+10\Gamma'_A$	Subcase 3b	
	$(4M+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{SK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{TK}}{\varsigma}\right.+\left\lceil\frac{\overrightarrow{AMA}}{\varsigma}\right.)\Gamma'_M$ $+(4M+7)\Gamma'_A$	Subcase 3c	
DA'_1	Γ'_M	Normal	N/A
	Γ'_M	Subcase a	
	$M\Gamma'_M$	Subcase b	
	$M\Gamma'_M$	Subcase c	
DA'_2	$2\Gamma'_M+2\Gamma'_A$	Normal	B-19
	$2\Gamma'_M+2\Gamma'_A$	Subcase a	
	$M[2\Gamma'_M+2\Gamma'_A]$	Subcase b, c	
S'_1	$(\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)\Gamma'_M+3\Gamma'_A$	Find First/all	B-20
S'_2	$(\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK})\Gamma'_M+2\Gamma'_A$	Find Min/Max	B-20
S'_3	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\left\lceil\frac{\overrightarrow{PK}}{\gamma}\right.)\Gamma'_M+3\Gamma'_A$	Find First/all	B-20
S'_4	$(1+\left\lceil\frac{\overrightarrow{LN}}{\gamma}\right.+\overrightarrow{PK})\Gamma'_M+2\Gamma'_A$	Find Min/Max	B-20

Table B-4. Algorithm Timings (page 5)

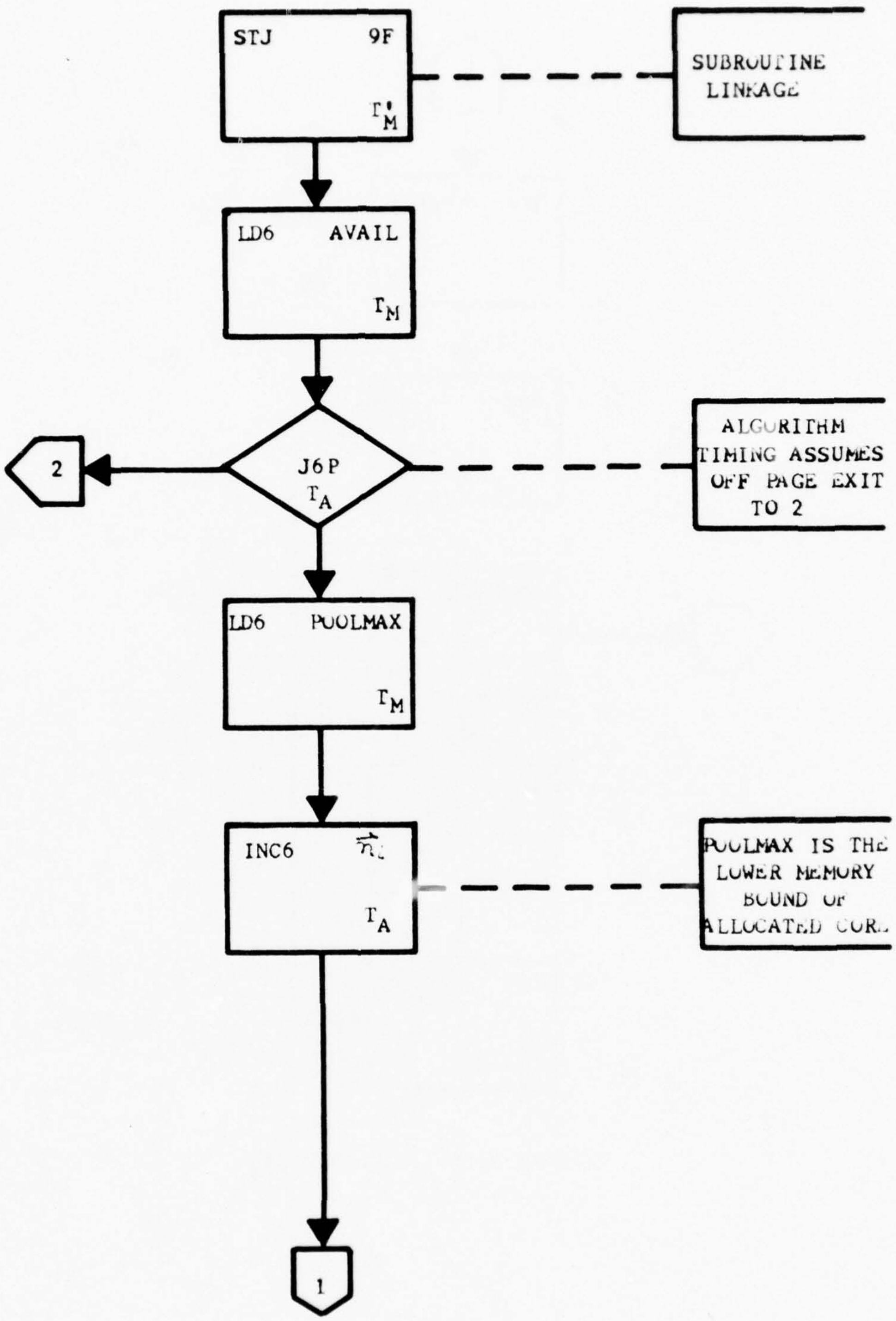

STJ 9F
T'_M
SUBROUTINE LINKAGE
LD6 AVAIL
T_M
J6P
T_A
2
ALGORITHM TIMING ASSUMES OFF PAGE EXIT TO 2
LD6 POOLMAX
T_M
INC6 $\vec{n}_i$
T_A
POOLMAX IS THE LOWER MEMORY BOUND OF ALLOCATED CORE
1

Figure B-1A. Algorithm A_1

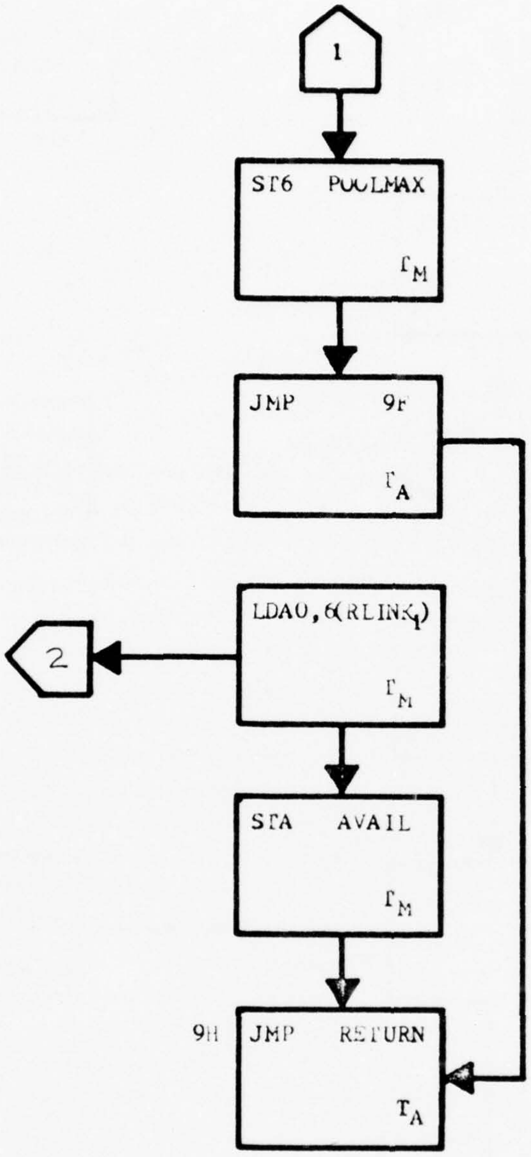


Figure B-1B. Algorithm A_1

a stack of available nodes maintained in a singly linked list called AVAIL. The timing corresponds to the case where all available storage is placed in AVAIL at the outset. It represents the shorter time case and is the one that will be used in the comparisons.

Algorithm DA_1

Algorithm DA_1 deallocates a node after it has been unloaded. Deallocation is independent of node use and simply places the node at the top of the AVAIL stack. It is shown in Figure B-2.

Algorithm I_1

Algorithm I_1 (Figure B-3) inserts a node represented by register six to the left of node rI_1. This means that a node is somehow determined and its address inserted in register one. The most common means of doing this is to enter register one with a list name in the case of a queue (FIFO). This algorithm then inserts the current node at the left end of the queue. For this reason the dotted enter block is included. This same algorithm can be used for insertion into a queue (LIFO) by changing the link field specifications in the instructions.

Algorithms I_2 and I_3

Algorithm I_2 is used to insert a node into a priority queue. The only difference between it and algorithm I_3 is that the latter algorithm includes a priority field that is to be used as part of the sort-in process. The priority algorithm is shown embedded within I_2 by the addition of the dotted steps. These steps are simply eliminated for I_2. Another way to look at these algorithms is to

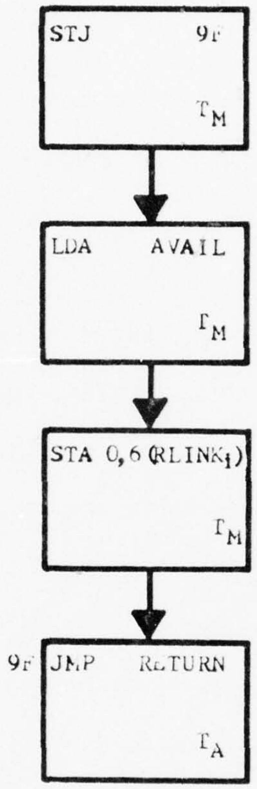


Figure B-2. Algorithm DA$_1$

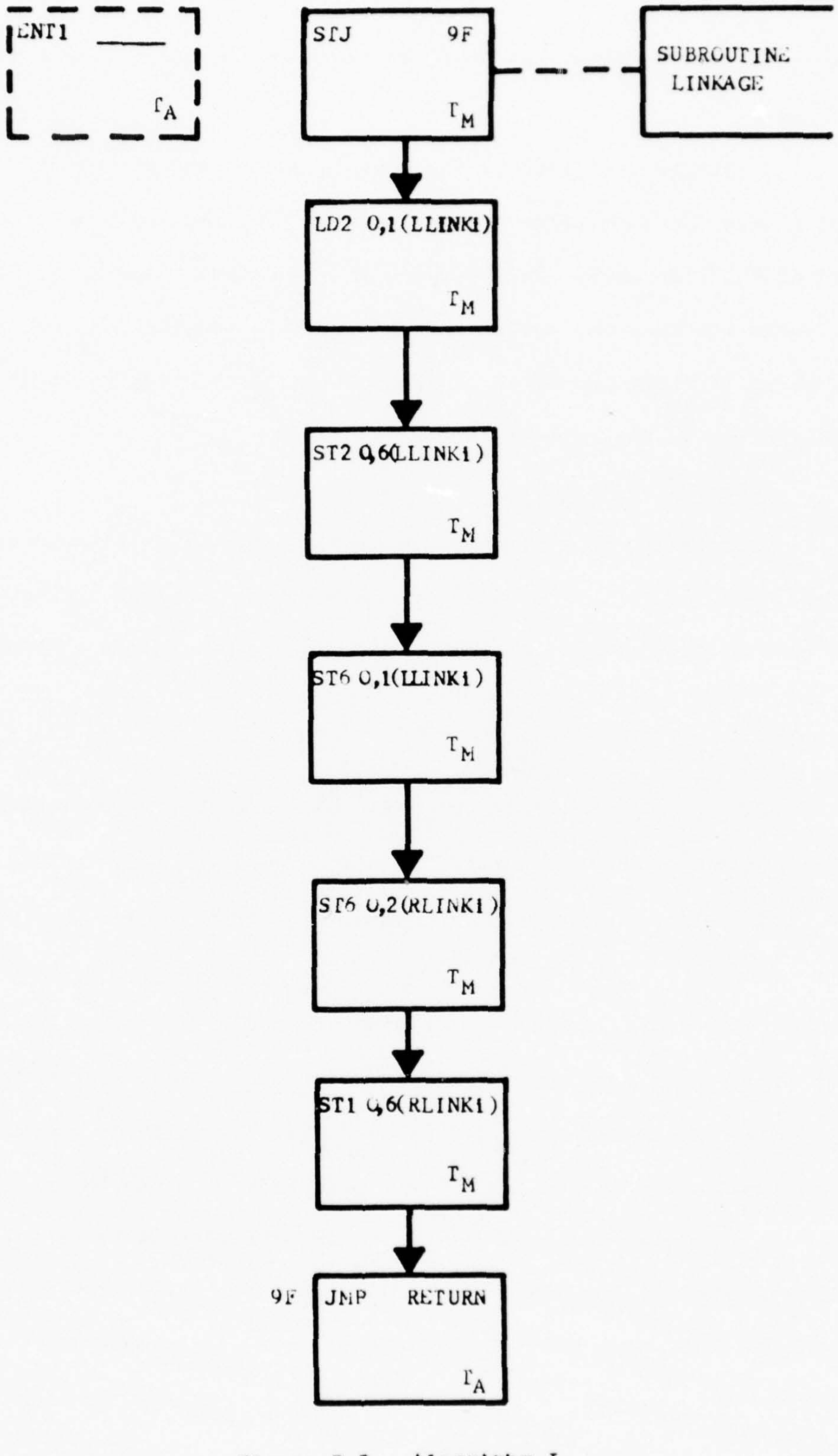


Figure B-3. Algorithm I_1

consider them an extension of I_1 by the addition of the sort-in process.

I_2 (Figure B-4) inserts the node in such a manner that if two nodes have the same value for the sort-in key, the new node is placed behind the list node. This results in the default ranking of FIFO. I_3 works the same way except that the default ranking does not take place until after the first ranking on the sort-in key followed by the ranking on the priority key.

Algorithms $\underline{D_1}$ and $\underline{D_2}$

Algorithm D_1 in Figure B-5 is used to delete the first node of a selected list. The only difference between D_1 and D_2 in Figure B-6 is that D_1 includes a list empty check, since D_2 assumes a prior search.

Algorithms $\underline{S_1}$, $\underline{S_2}$ and $\underline{S_3}$

These three algorithms comprise the searches that will be used for comparison. S_1 and S_2 are shown in Figure B-7. S_1 is intended to find the first list member whose search key satisfies the comparison criterion. The search is based on the priority queue, case four, as mentioned earlier under the discussion about algorithms in Chapter III. S_2 is used to find all the list nodes whose search key satisfies the comparison criterion and then to tag these nodes. The additional instructions for S_2 are shown in the dotted boxes.

S_3 in Figure B-8 is designed to find the maximum or minimum value among the nodes of a random list based on the search key. These searches were singled out with a separate designator because

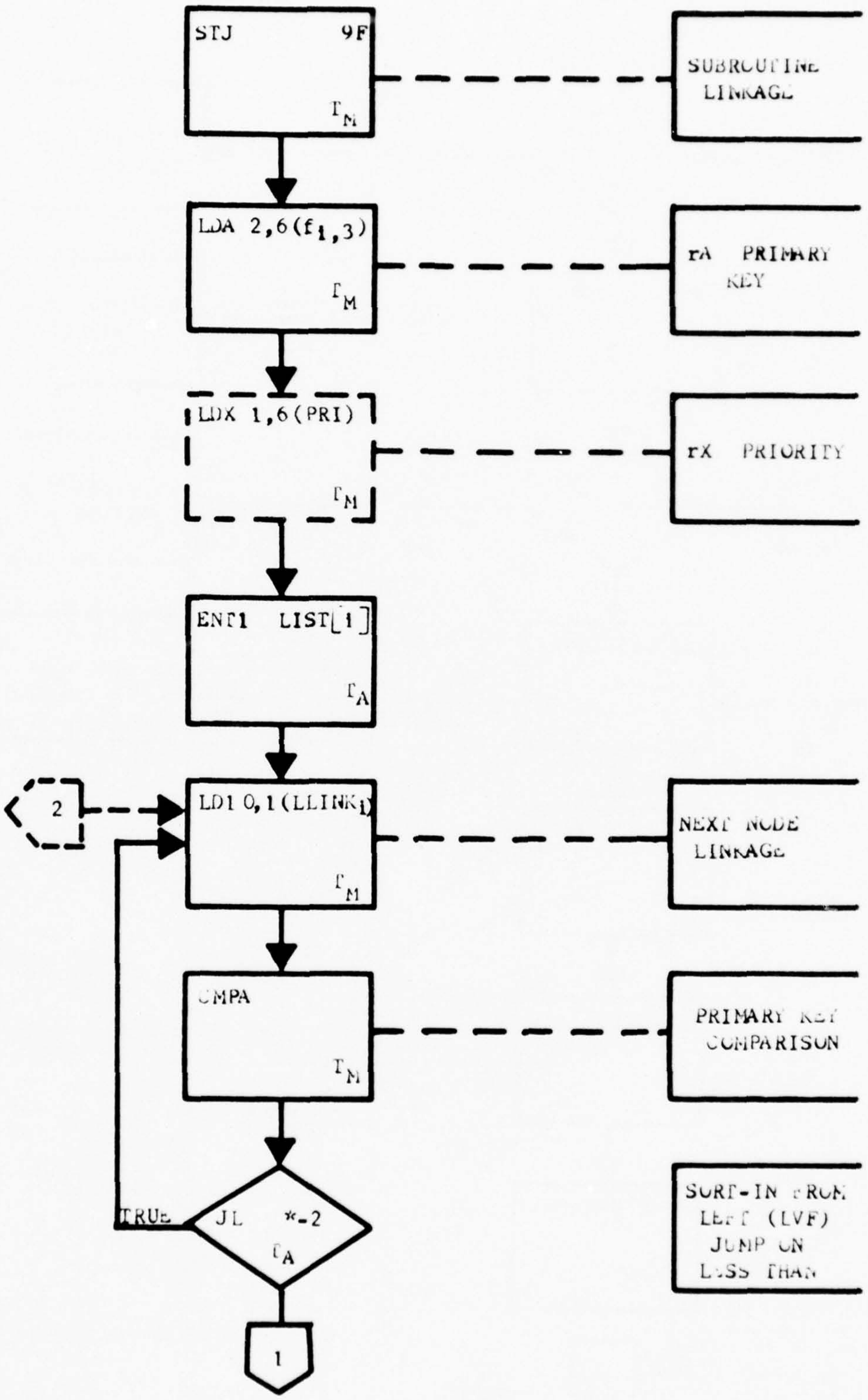


Figure B-4A. Algorithm I_2, I_3

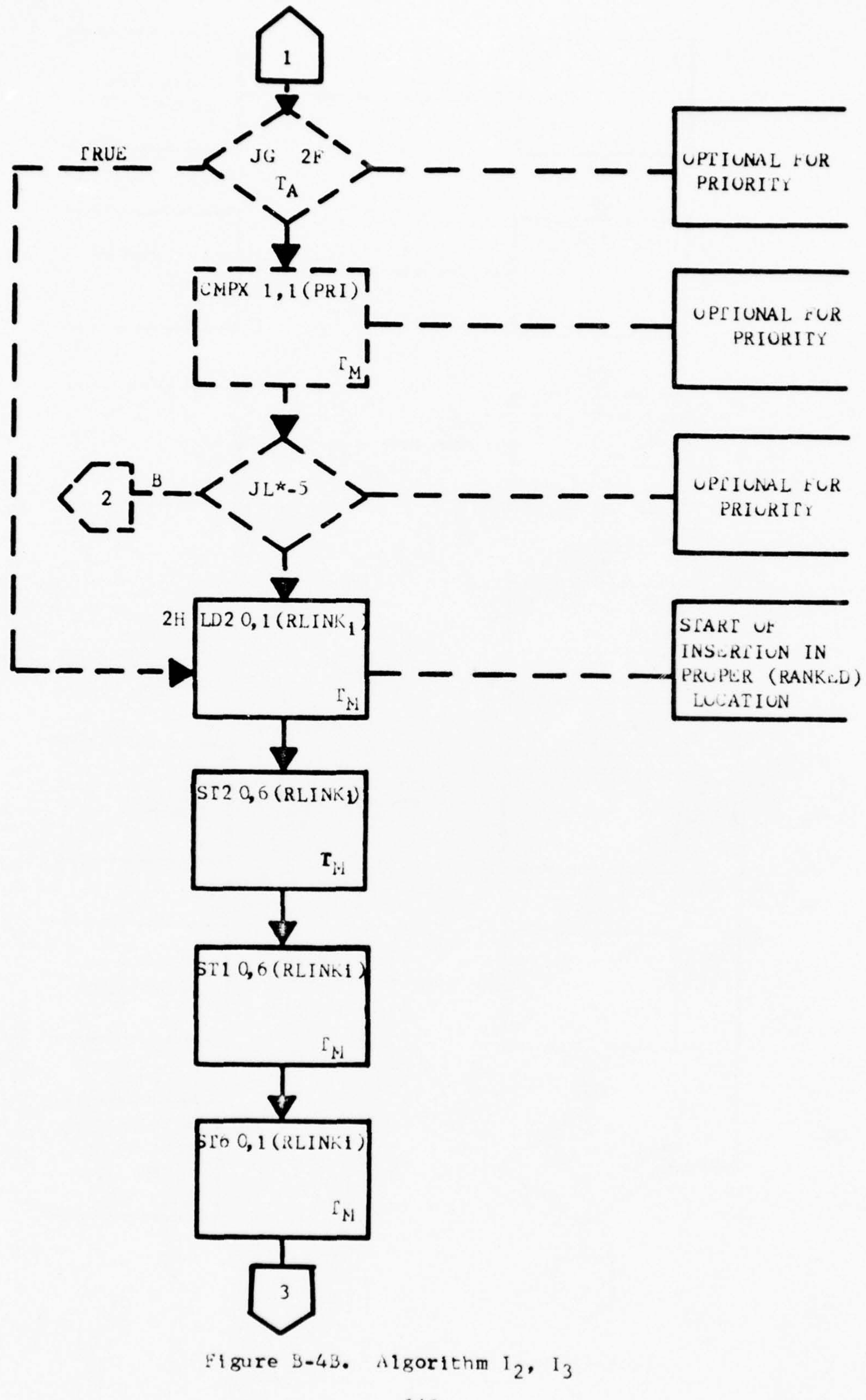


Figure B-4B. Algorithm I_2, I_3

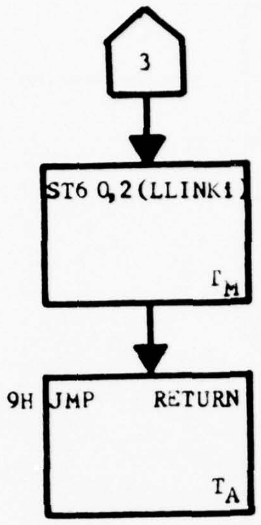


Figure B-4C. Algorithm I_2, I_3

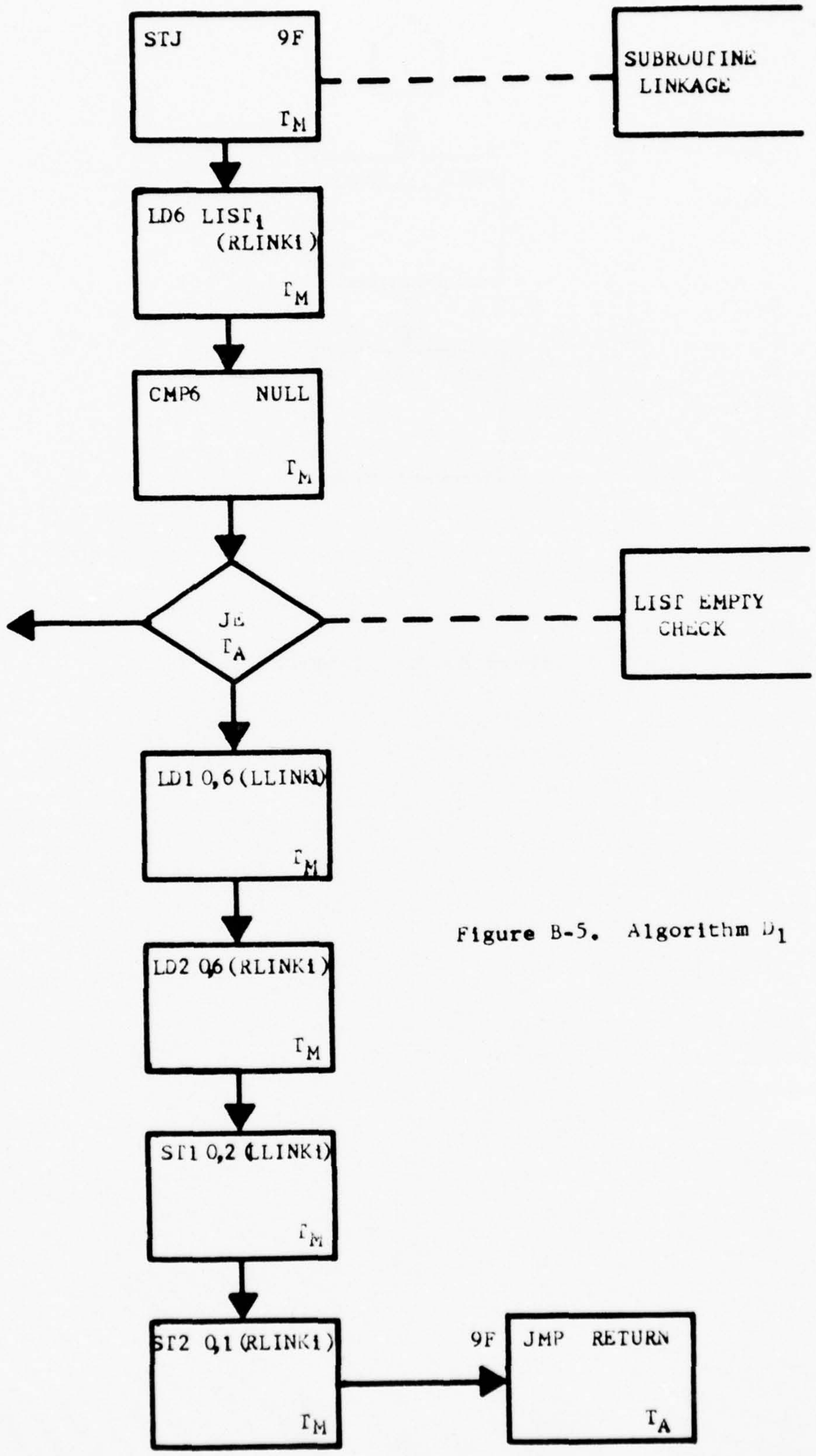


Figure B-5. Algorithm D_1

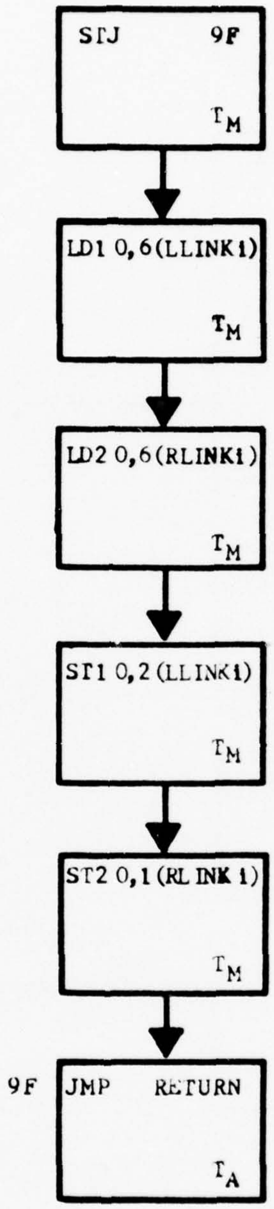


Figure B-6. Algorithm D_2

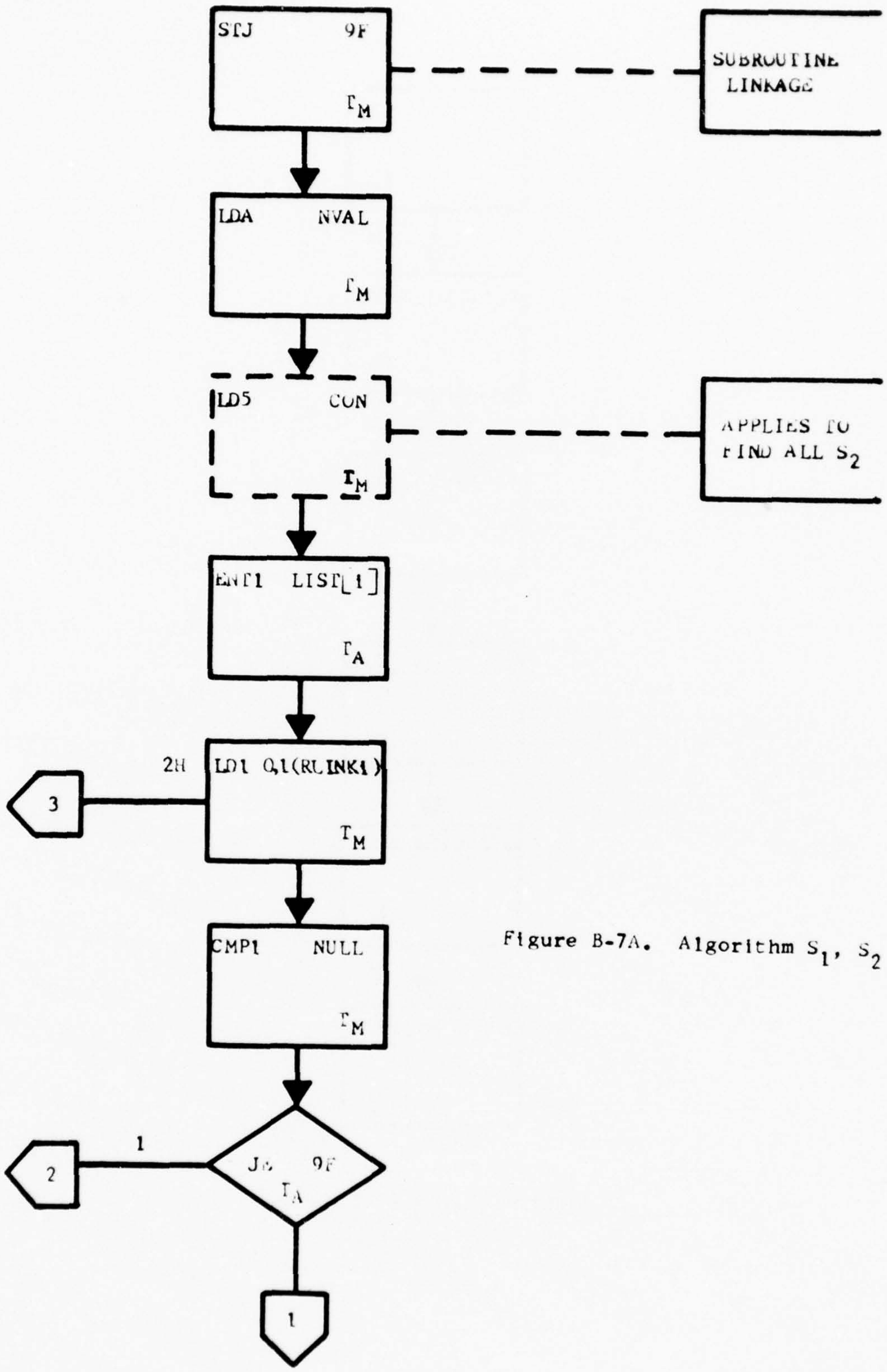


Figure B-7A. Algorithm S_1, S_2

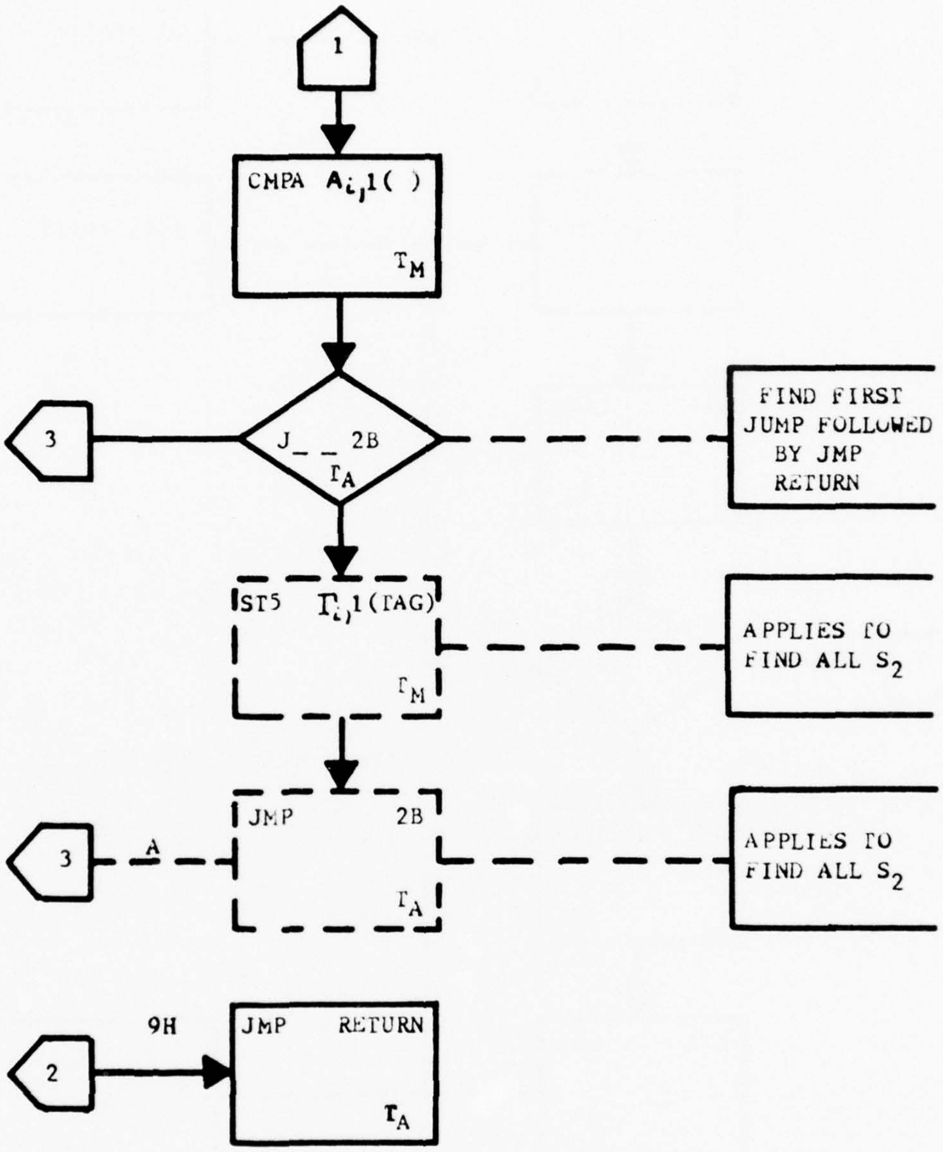


Figure B-7B. Algorithm S_1, S_2

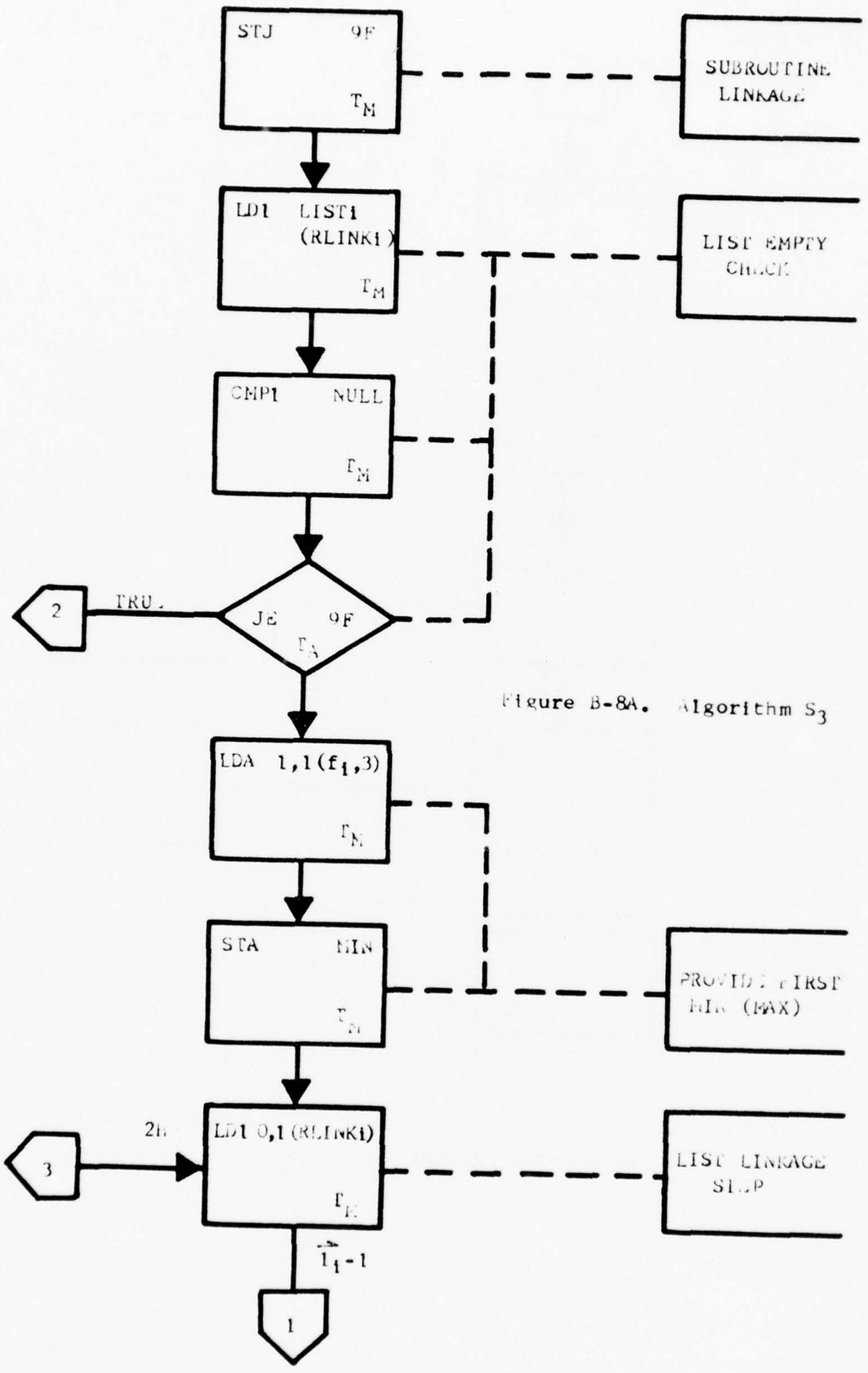


Figure B-8A. Algorithm S_3

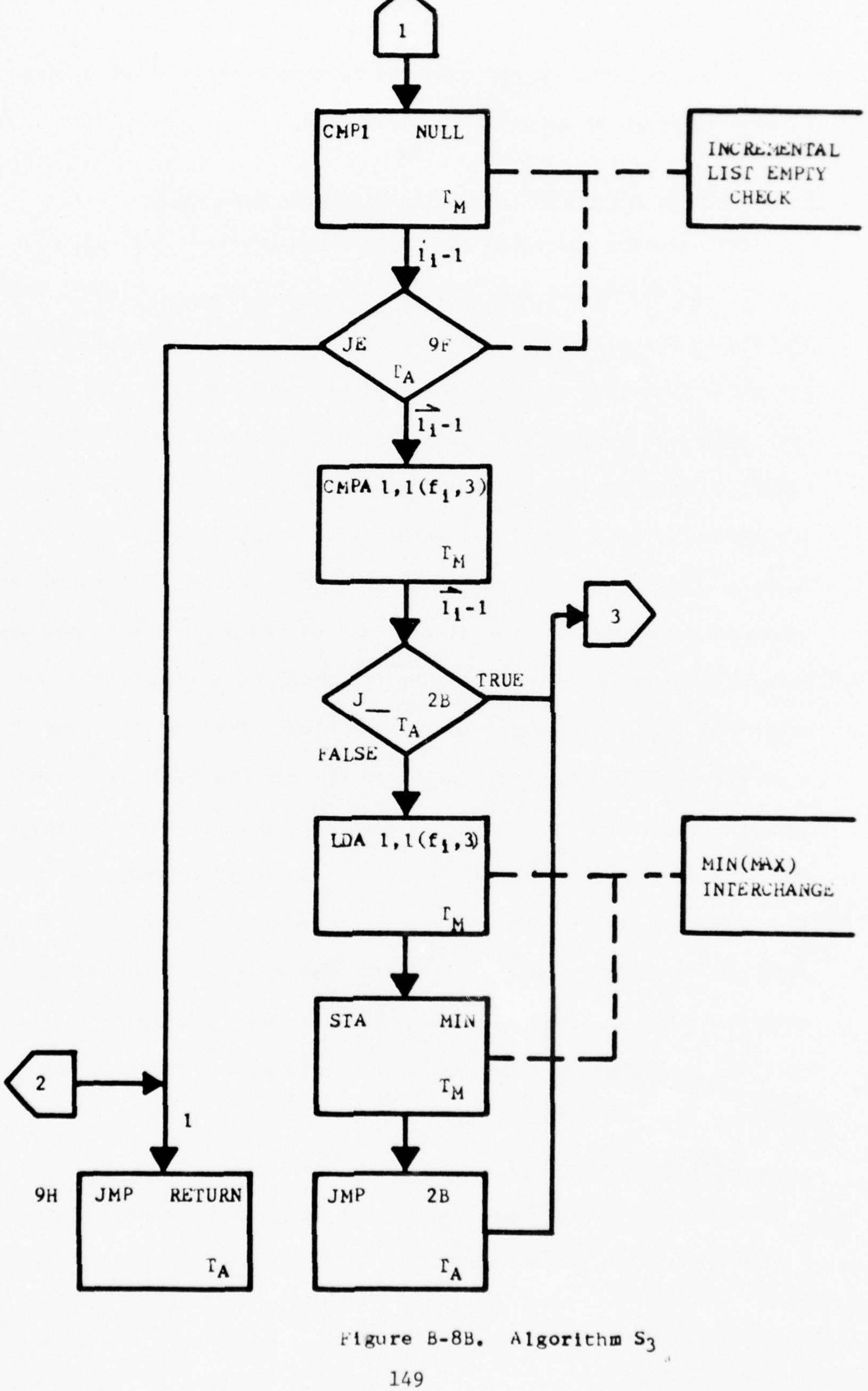


Figure B-8B. Algorithm S_3

as will be seen they do not have the data parallelism that S_1 and S_2 have in terms of associative processing.

B.3. MIX-AM, MIX-AM/RAM, and MIX-AM/RAM/AML Algorithms

This section describes the following algorithms: A_1', A_2', A_3', I_1', I_2', I_3', D_1', D_2', D_3', D_4', D_5', D_6', D_7', D_8', D_9', D_{10}', D_{11}', DA_1', DA_2', S_1', S_2', S_3' and S_4'. Before individual descriptions of the algorithms are given there are some general remarks that apply to all algorithms. Each algorithm assumes the storage structure discussed in Chapter II, Figure 4, appropriate to the particular memory organization. Times are shown for each individual instruction, with the exception of Lewin's algorithm, as noted later. The only difference between the instructions used for these algorithms and those for the random access memory is in the search or compare commands. The associative commands can deal with several words at a time. They are followed by a single or double slash depending on whether the response store register should be initialized or left with the previous result, so that the results of the next compare can be concatenated with the last compare. At the conclusion of the compare command it is assumed that the address of the first (lowest number memory location) is made available in index register one, since such a compare is always followed by a test on index register one to see if the list was empty.

Algorithms A_1' and A_2'

The allocation schemes for the AM and AM/RAM memory configuration are straightforward; however, some background is needed on the configuration of the memory system prior to the start of the simulation. Before the simulation starts, a one-time memory setup similar to that used for the random access memory is established. In the RAM case all available memory core was linked together based on the number of words necessary for each node. (Keep in mind that uniform node width is used throughout the research.) In the associative cases a determination is made as to how many words per node are necessary for the AM alone or how much for the AM and how much for the AM and the RAM in the combined configuration. A process is then undertaken which places within each AM node a zero in the first word busy bit location and an X (based on three state logic) in each node word busy bit location other than the first word. Further, the random access memory node that is to serve as the companion or buddy to the AM node has the location (address) of its first word stored in the RAM node address field (RNA) of the appropriate AM word (see storage structure in main body of text). In this way a search on the AM immediately reveals the address of the buddy RAM node in the combined configurations. This is done by the fact that the last step of any compare in the AM is to place the address of the first (lowest address) responder in index register one if there are any responders and a minus zero if there are none. The normal MIX indexing then permits access to the RNA since the AM can be addressing in parallel or by word.

For allocation (algorithms A_1', A_2', Figure B-9), which is done by a minimum compare on the busy bit, the first available AM node

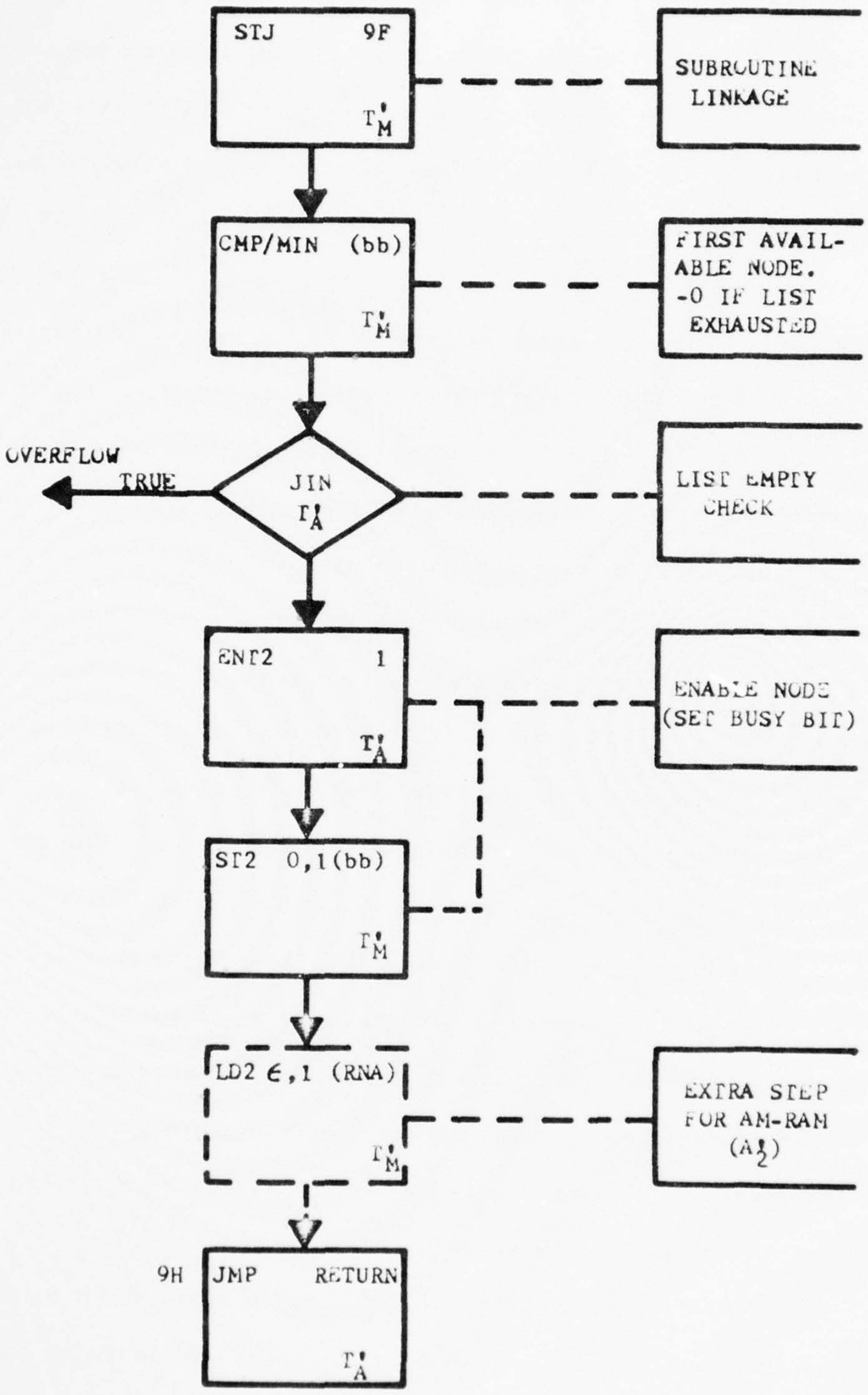


Figure B-9. Algorithm A'_1, A'_2

address is placed in index register one. An extra step is required to place the RNA into index register two for the combined memory case. During any initial compare (single slash) the X states in the busy bit lock out those words, acting as a busy bit zero. This is also used in general throughout all algorithms since the list name field stored in the first AM word is always queried first, and this is the word that always has a zero or one in the busy bit for normal AM operation. For a follow-on concatenated search (double slash) the X states act as a one in the busy bit location. The balance of the algorithm uses standard MIX commands to store a one in the busy bit location of the allocated node, and this is followed by a return jump.

Algorithm A_3

This algorithm is used to support the MIX-AM/RAM/AML memory organization. It is somewhat more complicated than the others because of the particular way the delete algorithms based on Lewin's algorithm operate. The following discussion is based on Figure B-10.

The first instruction provides for the subroutine linkage. The next instruction compares register X with the value ST_2. Consider that there is a variable called ST_4 stored in register X and therefore the compare command compares ST_4 to ST_2. The outcome of the compare as tested by the JLE command determines whether the new node is to be drawn from the AM or the AML. To understand the physical meaning of making this choice consider that this algorithm is only used for priority queues and for entries of new schedulable

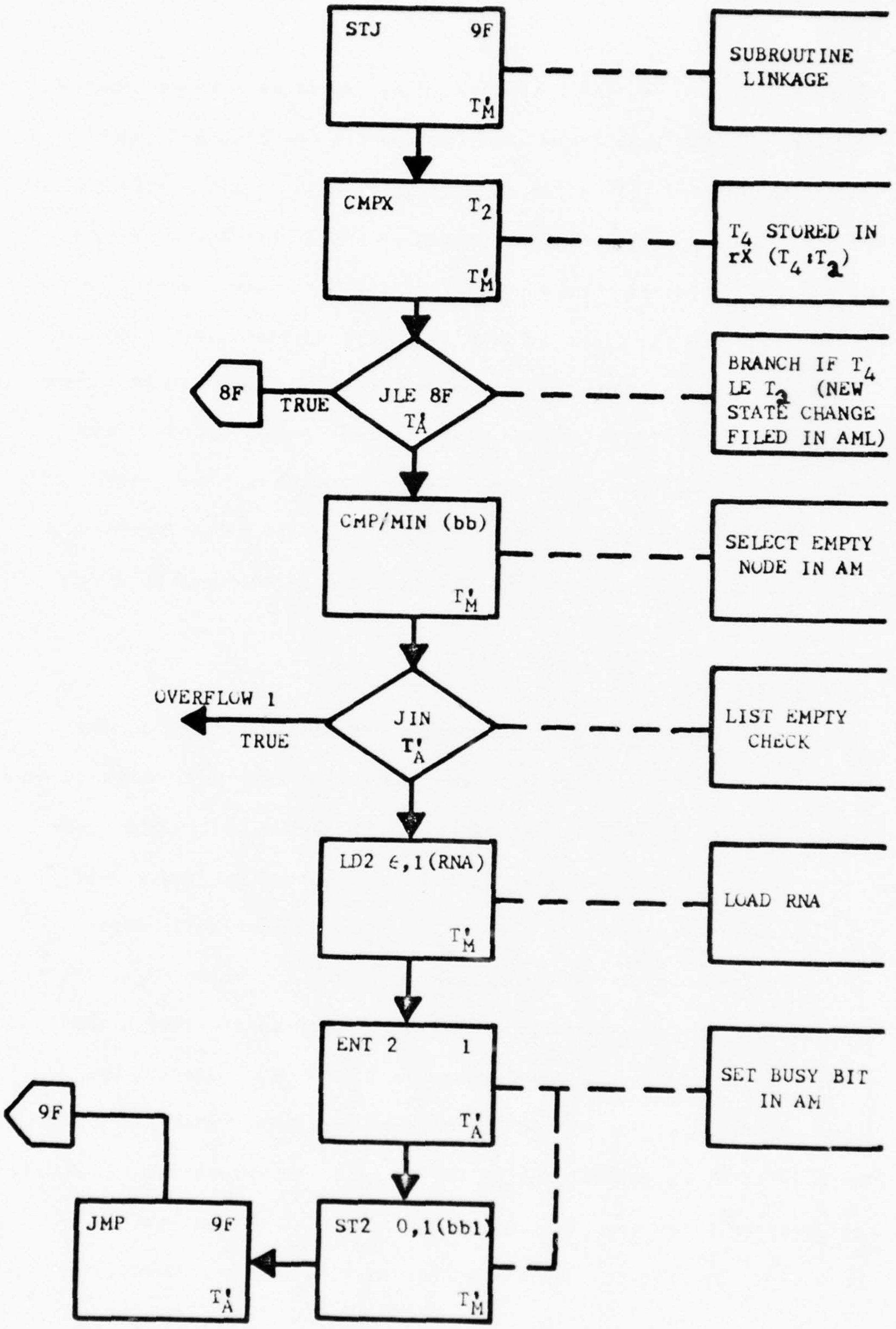


Figure B-10A. Algorithm A_3'

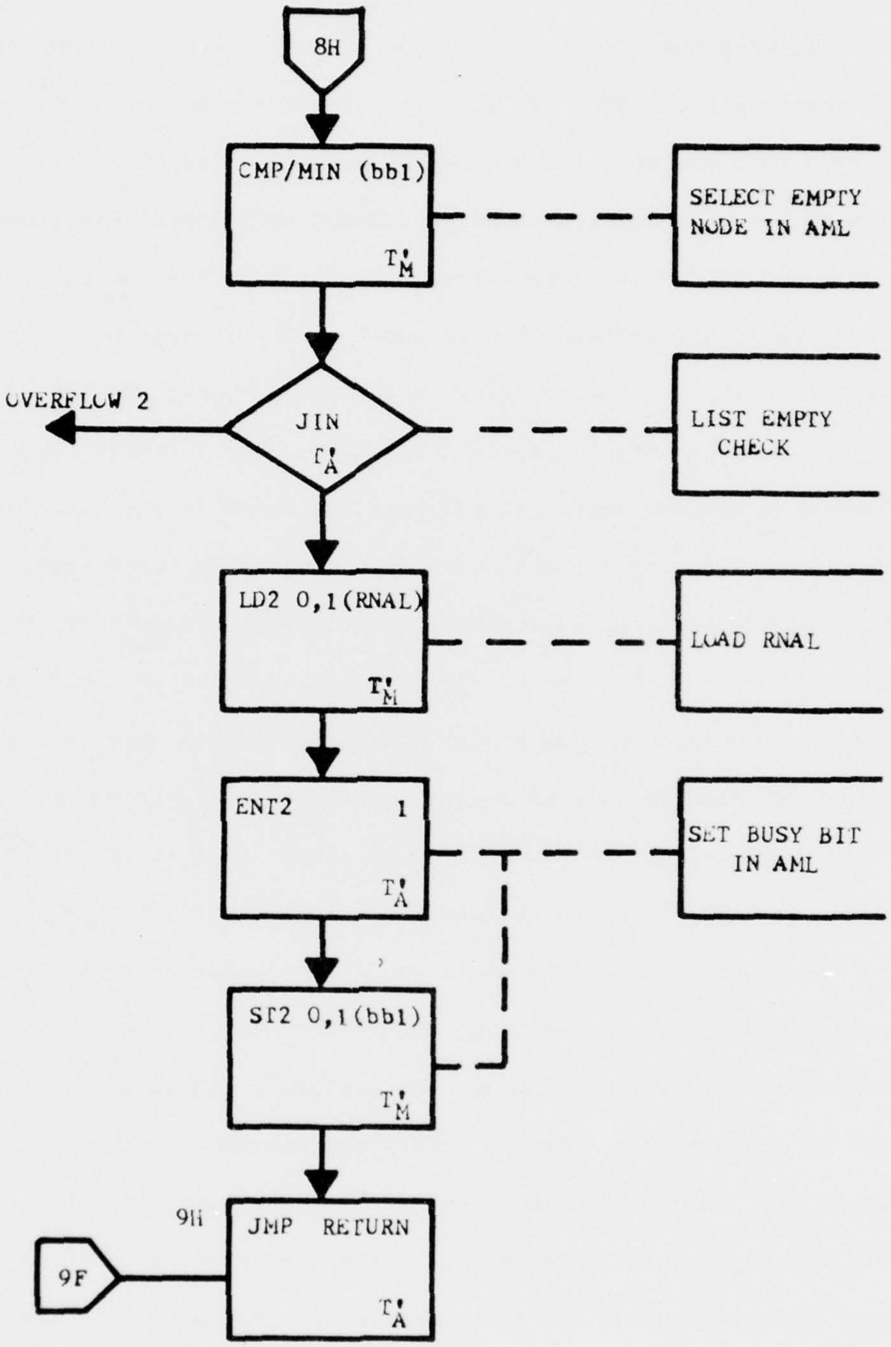


Figure B-10B. Algorithm A'_3

state changes that result from a state change selected from the AML. ST_2 represents the greatest time or primary key of any state change entry within the AML, and ST_4 represents the state change time or primary key of a new state change spawned by a state change notice within the AML whose state change time was less than or equal to ST_2. Therefore if ST_4 is less than or equal to ST_2 it must be filed in the AML to avoid a timing error in the state change process.

Assume that ST_4 is greater than ST_2. Then a simple allocate based on a minimum search of the busy bit field (bb) is made, followed by a list empty check based on the entry to index register one. If the register is negative, indicating no more available nodes, then an exit is made. If there is at least one available node the companion random access node address (RNA) previously established for each AM node is loaded into index register two and the busy bit is set to one, indicating an active node by the enter and store commands, which is followed by a jump return.

In the alternate case where ST_4 is less than or equal to ST_2, an allocate process takes place within the AML. At this point the buddy system between the AM and the auxiliary RAM core can not be used to store this new entry. Therefore the AML has stored within one of its fields a random access memory node (set aside for each word within the AML) whose address is transferred to index register two. In this way the AML functions in the same manner as the AM with regard to auxiliary RAM core. The balance of the algorithm deals with setting the busy bit within the AML (bbl) followed by the jump return.

Algorithms I_1' and I_2'

These algorithms are shown in Figures B-11 and B-12, respectively. Algorithm I_1' is straightforward and is used to insert the list name into the selected node. There is an assumption made that not all operating data germane to a simulation is included in the composite node cycle timing. Only the data that is needed for node management for a particular memory implementation is included. As an example a list name is not needed explicitly for the RAM memory but is needed for the associative cases. On the other hand, linkage information is needed for the RAM case which is not needed for the associative cases.

The next algorithm, I_2', is a further example of the above comments. Here the additional information of time of entry (stored in register X) must be inserted for certain data cycles involving queues since physical location is used for the RAM but the AM requires some search parameter.

Algorithm I_3'

This algorithm, shown in Figure B-13, is used to insert the additional information needed to support the counter scheme (discussed in Chapter III) for maintaining queues. The major problem with using the associative memory is that a search or compare based on a minimum is not particularly parallel in the sense that there must be some reconciliation across the words and each word can not be treated in an independent fashion. Feng [21] has developed parallel

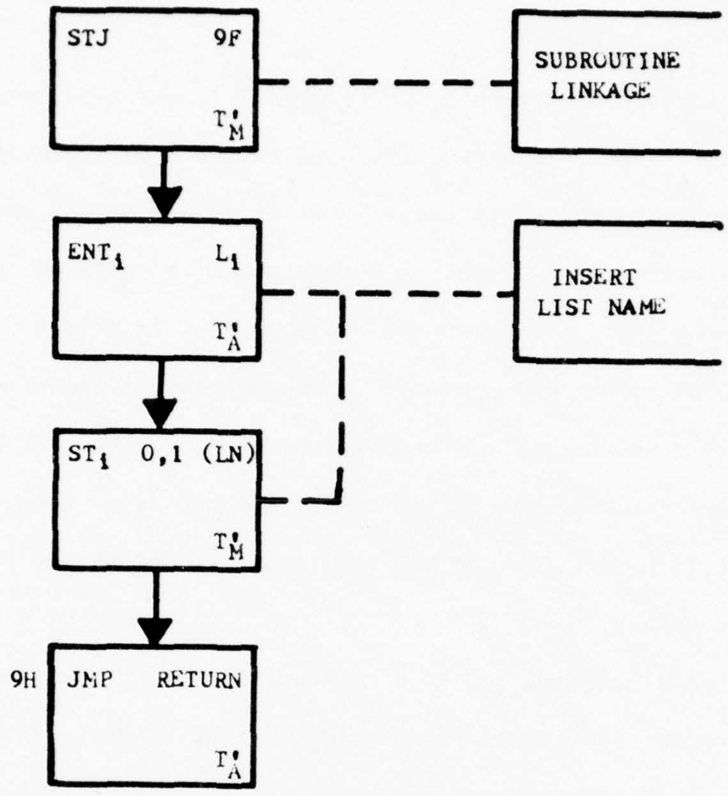


Figure B-11. Algorithm I'_1

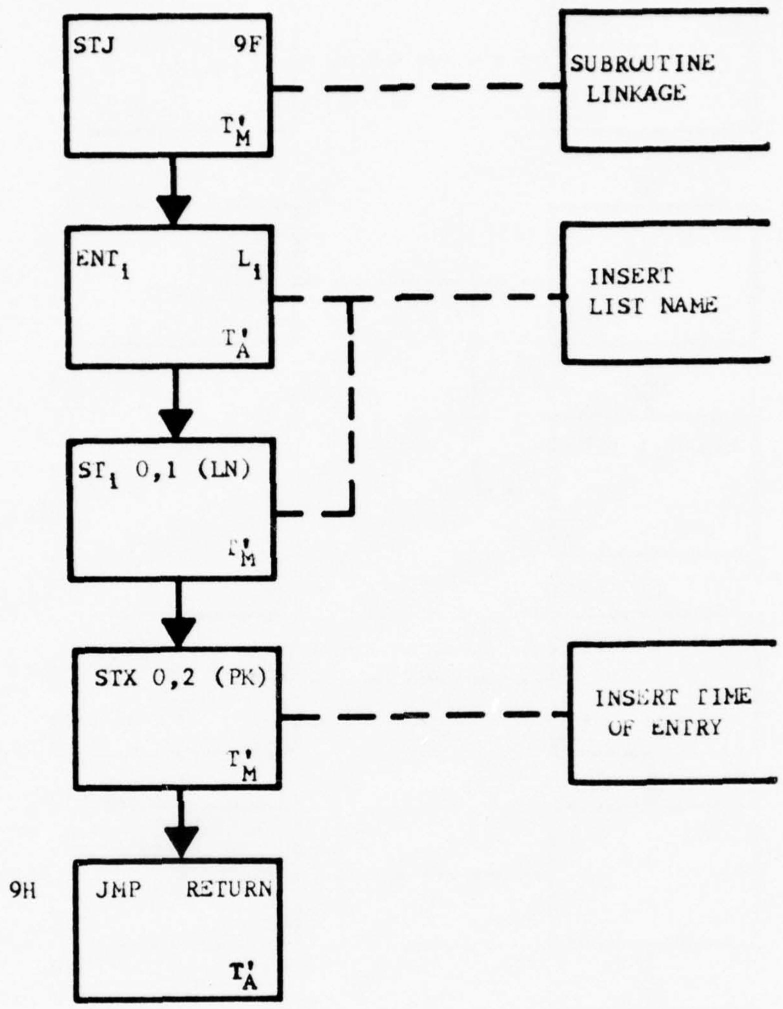


Figure B-12. Algorithm I'_2

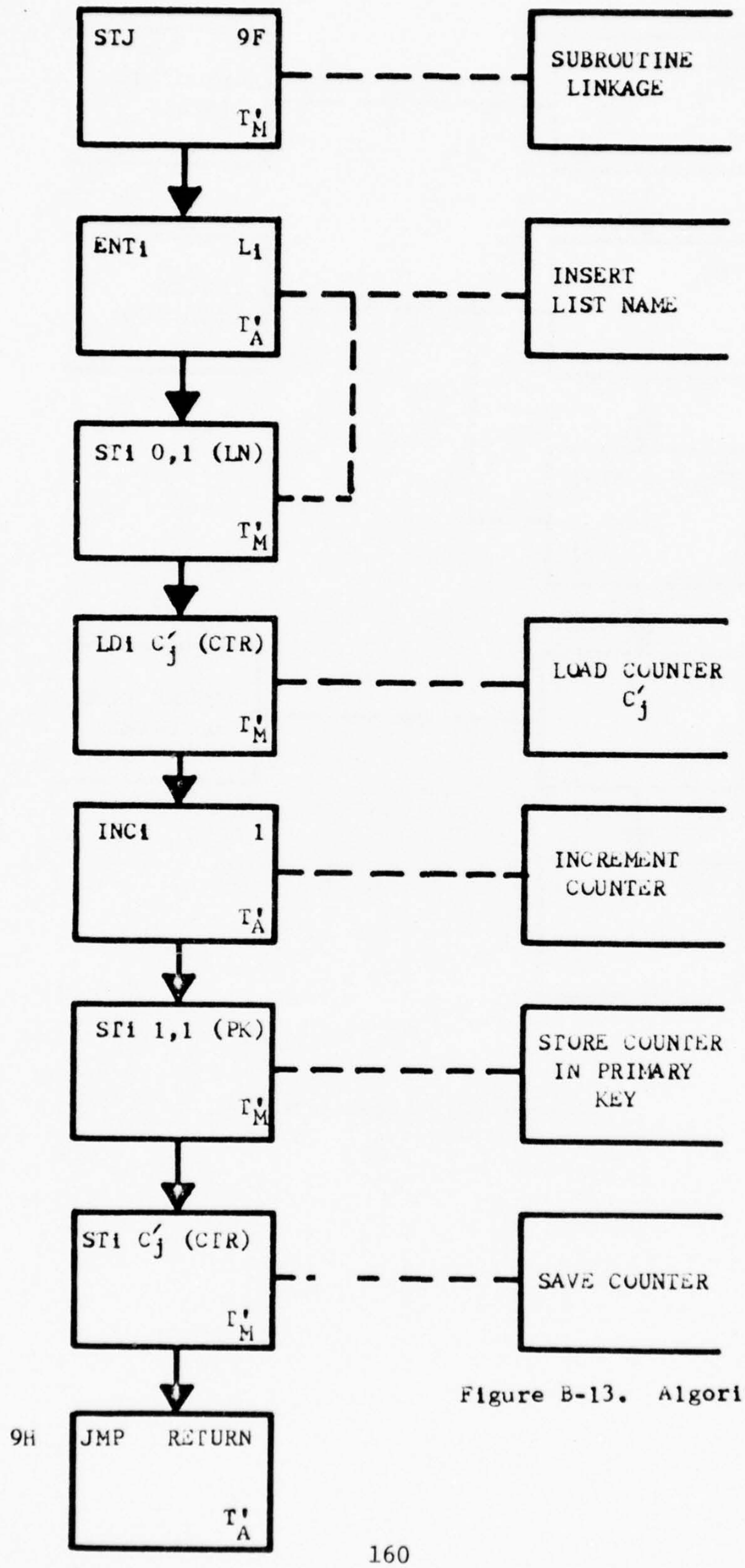


Figure B-13. Algorithm I'_3

minimum searches for an associative memory, but they are based on one bit slice at a time and therefore take w bit slices if the search field is w bits wide. One way out of this situation of linearly increasing search time with field search width is to attempt to convert from minimum search to a comparand search where each word may be treated independently and processing width can be increased advantageously beyond one. This situation is possible in queues such as the ones considered here because they are "pure" queues in the sense that entries are always made at one end of the queue and removals take place either at the same end or the opposite end. The entries are never removed from some intermediate location as they would normally be in the priority queue. For this reason counters can be used to serial number the entries as follows. For the LIFO queue, a counter maintains the last serial number given to any entry (largest value). When the last entry is to be removed (see delete algorithm) a search (comparand) on equal for the last value can be made and the single proper node retrieved. The counter is then incremented or decremented depending on whether an entry is going in or coming out. For a FIFO queue two counters are maintained, one for entry and one for removal.

This algorithm, as mentioned, adds the additional information to the associative node to make the counter scheme effective. Notice that there is more overhead for this scheme in the sense of more instructions than for minimum search; however, the additional time is more than made up during deletion with the processing width (γ)

greater than one. The algorithm starts with the subroutine linkage and then inserts the list name information. The counter is then loaded and incremented and stored in the proper associative node. A jump return completes the insertion process.

Algorithm $\underline{D}_1'$ and $\underline{D}_2'$

These algorithms are shown in Figure B-14. They complement algorithm I_3' in that they are the other half of the counter scheme for queues. The only difference between them is that D_2' has the extra step for the AM/RAM memory to load in the RNA. They start with the subroutine linkage which is followed by the selection of the list members by list name. A test on list empty is made, followed by the additional instructions necessary to support the counter. C_j' is used to denote the single counter in the LIFO case or the second counter in the FIFO case (see I_3'). The search is made on equal for the counter, after which the counter is decremented in the LIFO case and incremented in the FIFO case. The counter is then restored to memory, the RNA is loaded if required and a jump return is made.

Algorithms $\underline{D}_3'$, $\underline{D}_4'$, $\underline{D}_5'$, $\underline{D}_6'$, $\underline{D}_7'$ and $\underline{D}_8'$

This family of algorithms is shown in Figures B-15, B-16 and B-17. They are considered together because each pair can be considered to be a superset of the previous pair. Within each pair the odd number is for the straight associative case and the even for the AM/RAM case.

These algorithms form the basis for the straight implementation of the three versions of the priority queue considered in the

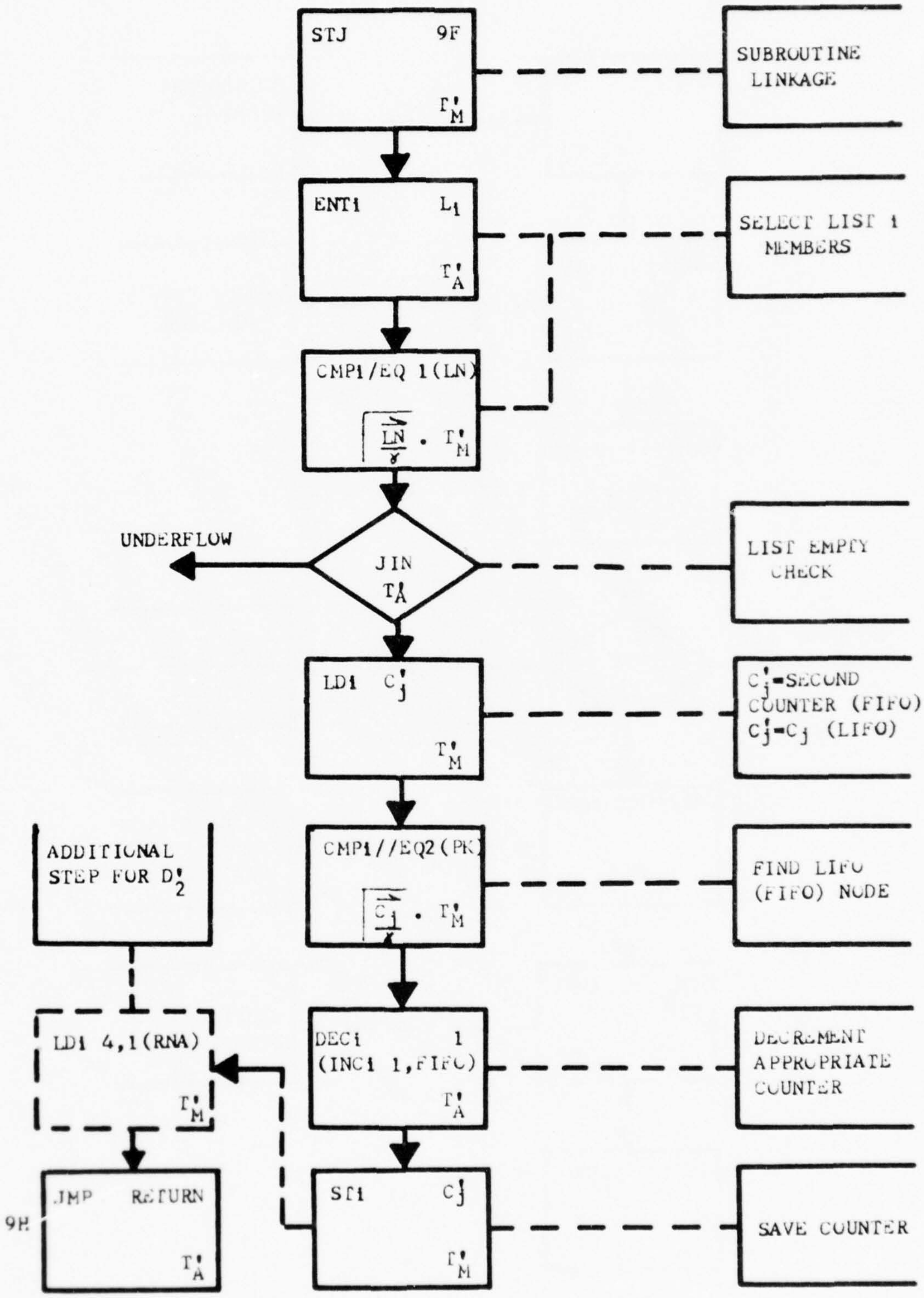


Figure B-14. Algorithm D'_1, D'_2

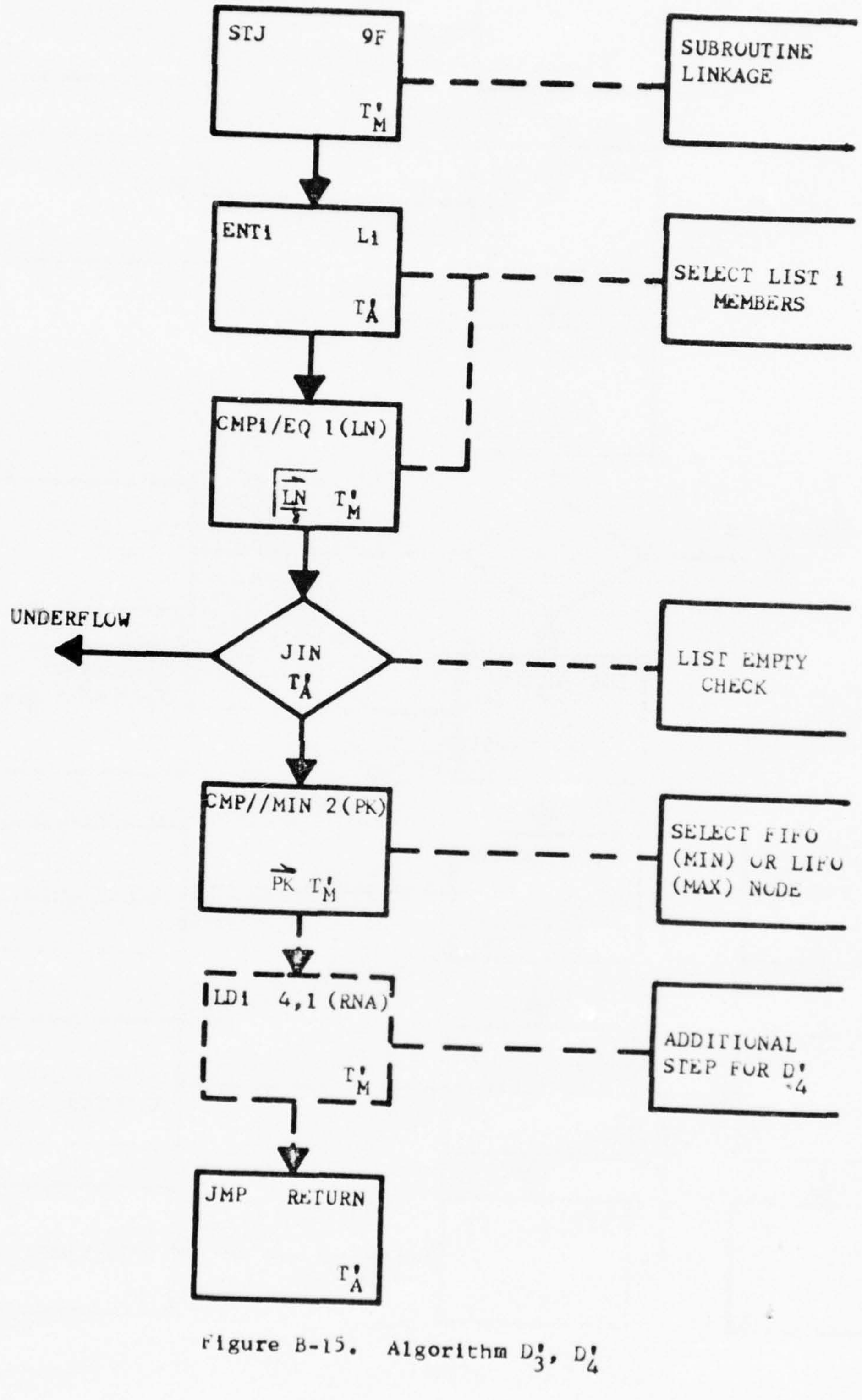
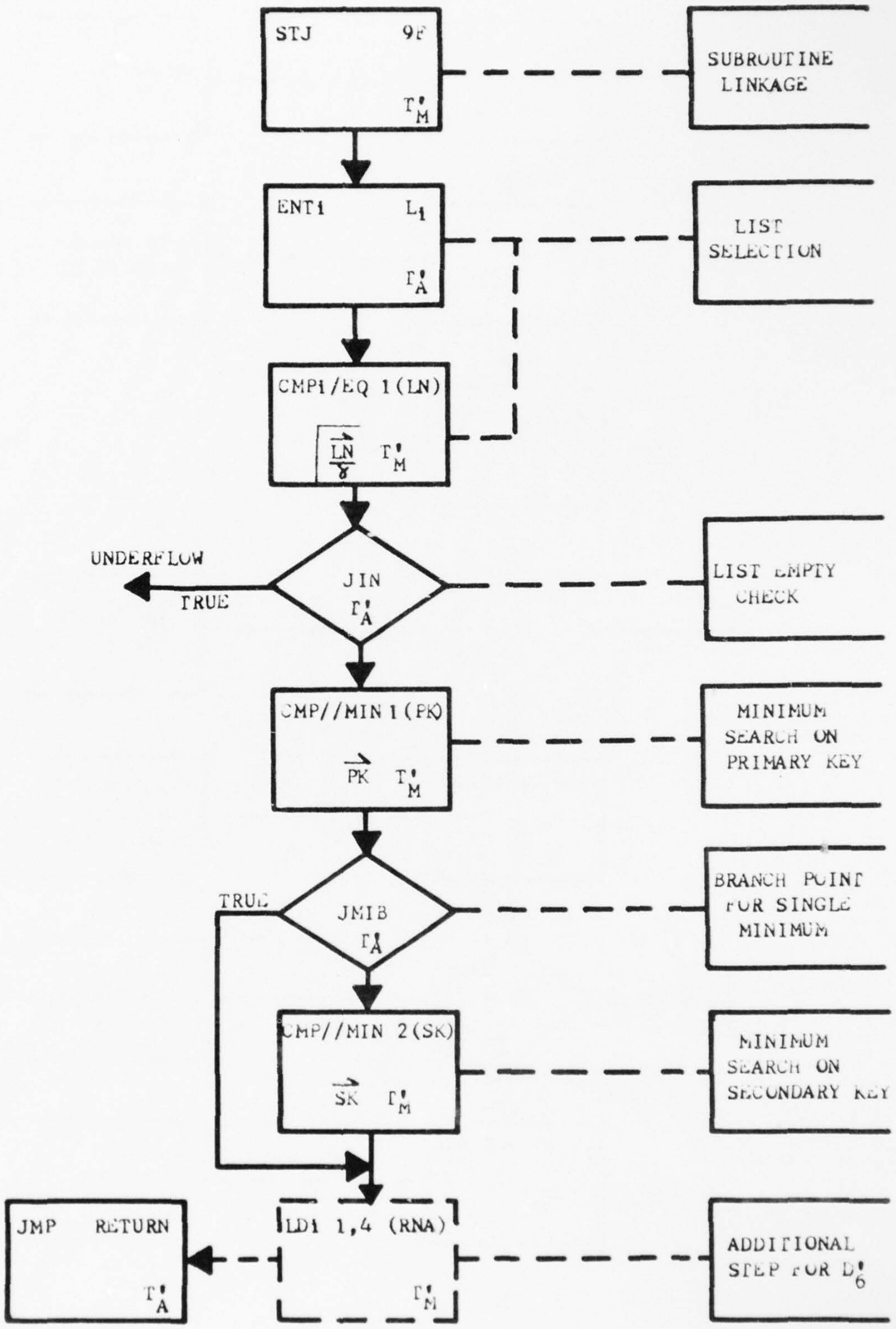
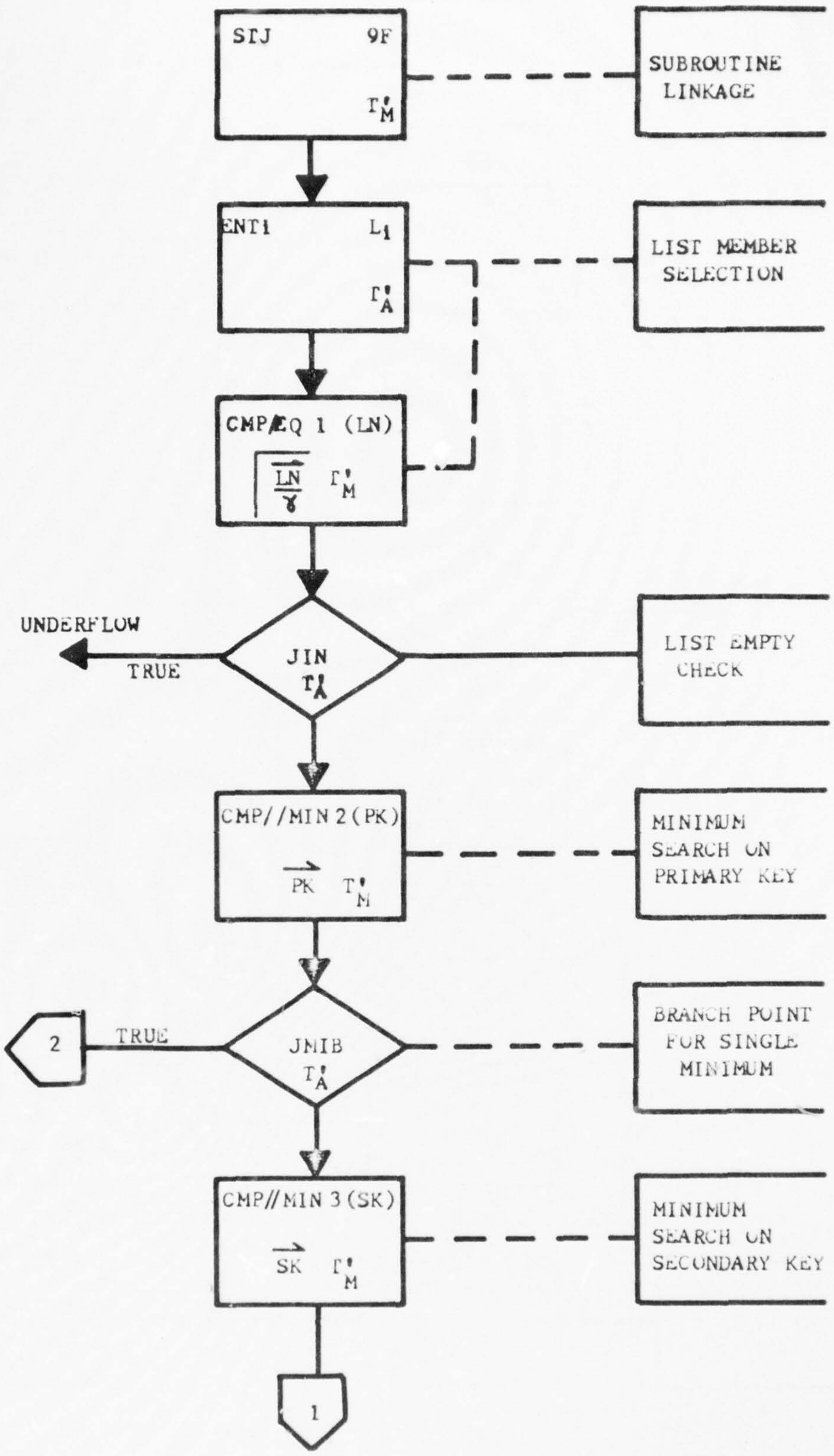


Figure B-15. Algorithm D_3', D_4'



STJ 9F
Γ'_M
SUBROUTINE LINKAGE
ENTi L_i
Γ'_A
LIST SELECTION
CMPi/EQ 1(LN)
$\overrightarrow{LN}$ / γ Γ'_M
UNDERFLOW
TRUE
JIN
Γ'_A
LIST EMPTY CHECK
CMP//MIN 1(PK)
$\overrightarrow{PK}$ Γ'_M
MINIMUM SEARCH ON PRIMARY KEY
TRUE
JMIB
Γ'_A
BRANCH POINT FOR SINGLE MINIMUM
CMP//MIN 2(SK)
$\overrightarrow{SK}$ Γ'_M
MINIMUM SEARCH ON SECONDARY KEY
9H JMP RETURN
Γ'_A
LDi 1,4 (RNA)
Γ'_M
ADDITIONAL STEP FOR D'_6

Figure B-16. Algorithm D'_5, D'_6



STJ 9F
Γ'_M
SUBROUTINE LINKAGE
ENT1 L_1
Γ'_A
LIST MEMBER SELECTION
CMP/EQ 1 (LN)
$\overline{LN}$ γ Γ'_M
UNDERFLOW
TRUE
JIN
Γ'_λ
LIST EMPTY CHECK
CMP//MIN 2 (PK)
$\overrightarrow{PK}$ Γ'_M
MINIMUM SEARCH ON PRIMARY KEY
2
TRUE
JMIB
Γ'_A
BRANCH POINT FOR SINGLE MINIMUM
CMP//MIN 3 (SK)
$\overrightarrow{SK}$ Γ'_M
MINIMUM SEARCH ON SECONDARY KEY
1

Figure B-17A. Algorithm D'_7, D'_8

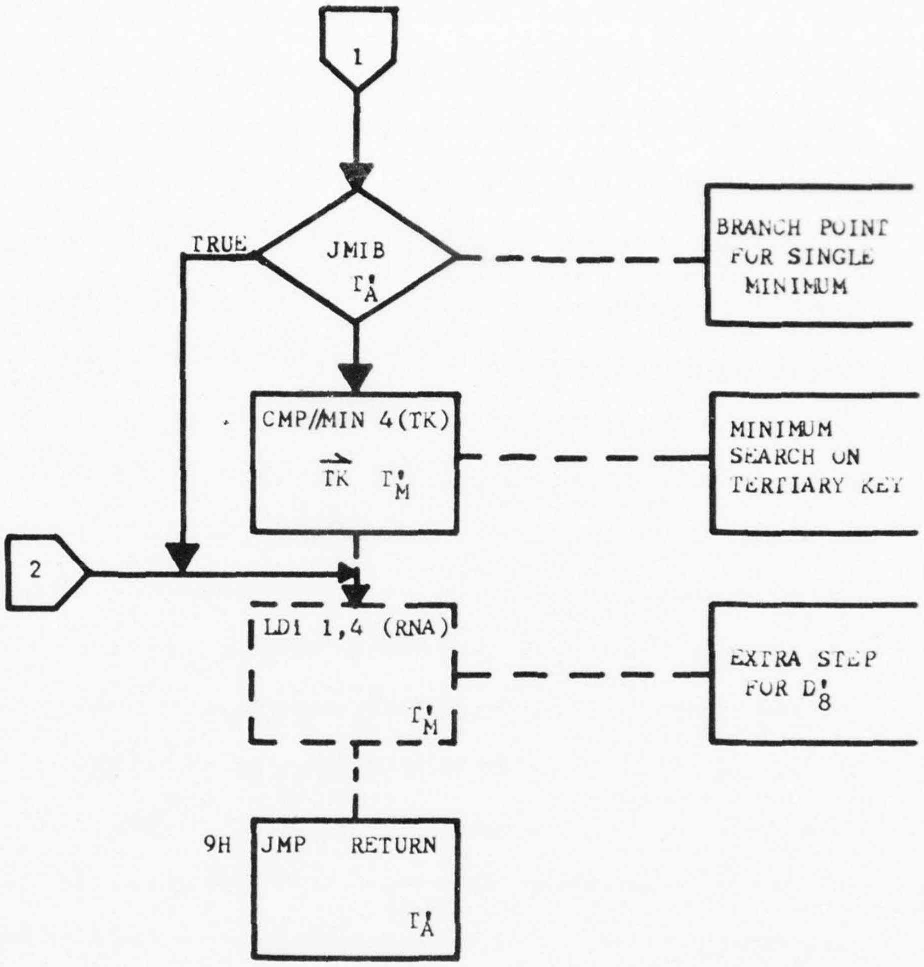


Figure B-17B. Algorithm D'_7, D'_8

research. In the first case there is a single primary key which is used to select the minimum node of the list, based on the primary key. In the second and third cases one additional key is added to form a joint primary key with the first primary key (PK). In this sense the secondary key, SK, and the tertiary key, TK, are not independent search keys as in a data management system. Their purpose in discrete simulation is to break ties for deciding the next state change. Both the SK and TK may be thought of as adding lower order bits to the PK to increase the resolution among state change notices in the priority queue. In general for discrete simulation PK may be thought of as the simulation time when the particular state change should take place, and the secondary key may be thought of as a priority key for tie breaking. TK exists implicitly within the RAM structure since the default ranking as part of the sort in process is FIFO. To maintain this same capability within the AM, TK must be explicitly present in the form of a time of entry simulation time. Therefore three keys are needed in the AM case, where only two are necessary in the RAM case--four if the list name field is counted. The major problem comes in when one considers that the priority queue must be maintained by a minimum search that increases linearly with search width. A possible solution to that problem is discussed later in conjunction with Lewin's algorithm.

All six algorithms follow the same pattern, so just the first two will be discussed. The algorithms start with the subroutine linkage followed by a selection of the list members and a list empty check. A search is then made on the primary key (note double slash for concatenated search with the list member selection). The first responder has its address loaded into index register one and if required the RNA is loaded also. A jump return completes the process. The remaining four algorithms simply add concatenated searches for SK and TK respectively. However, at the end of every intermediate search a test is made on the match indicator to determine if there is only one survivor. This is done because it is to be expected within a discrete simulation that the extra keys will not be needed for every data cycle and it would be a linear waste of time to use them.

Algorithms D'_9, D'_{10} and D'_{11}

These algorithms have been specially designed to alleviate the problem mentioned earlier in conjunction with priority queues--that is, the linear increase in time with increasing search width. The algorithms shown in Figure B-18 are based on Lewin's algorithm mentioned in Chapters III and IV. Lewin's algorithm has the particular property that one can guarantee an upper bound on the number of memory cycles necessary for ordered retrieval regardless of the field width, assuming contiguous bits. In terms of the research this means that a breakeven point exists between doing a straight minimum search and doing Lewin's algorithm. To utilize Lewin's algorithm, however, requires slightly different thinking from the straight node cycle concept designed to trace a single node through a birth and death process. This is because although one can make

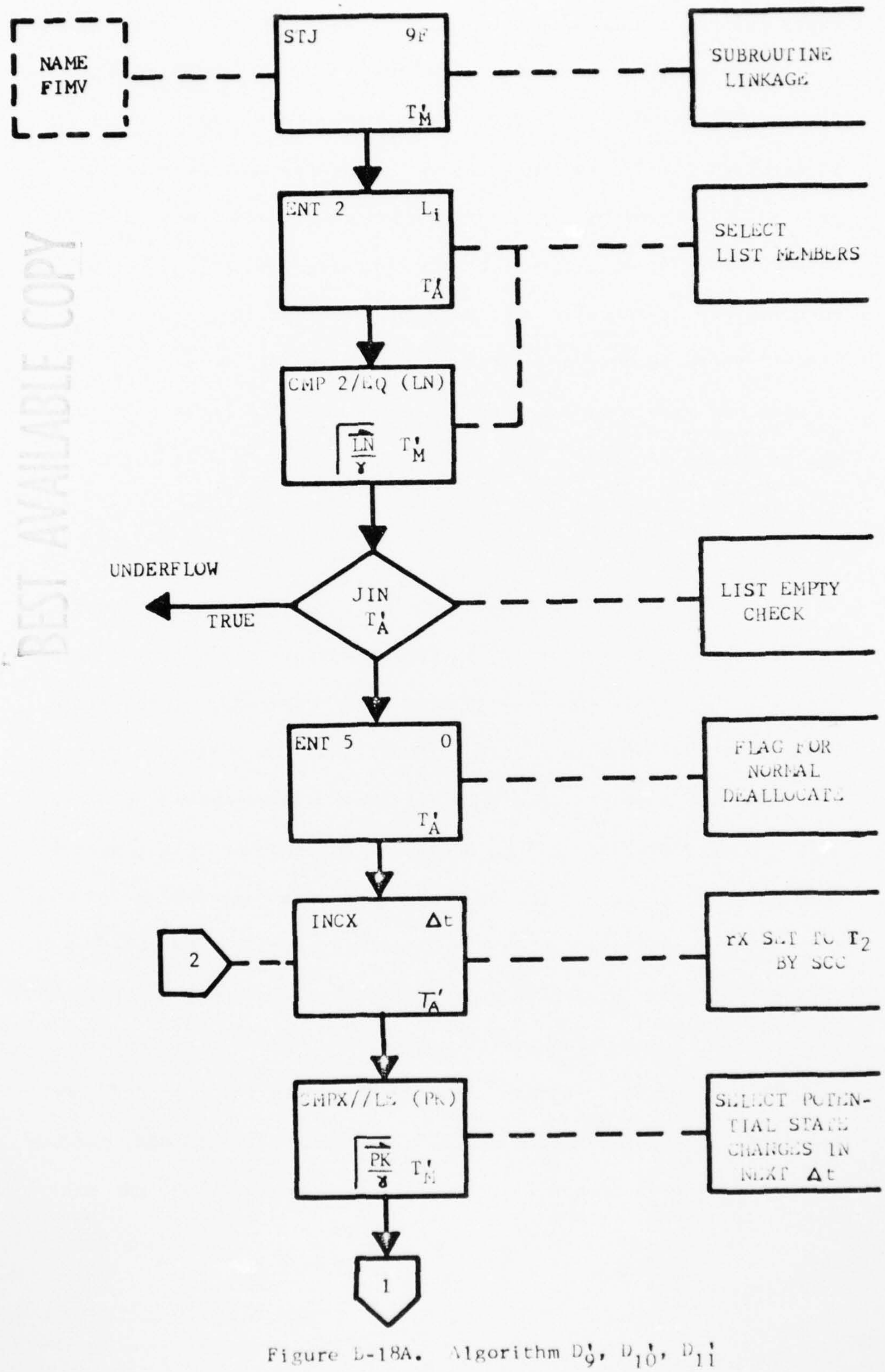


Figure D-18A. Algorithm D_9', D_{10}', D_{11}'

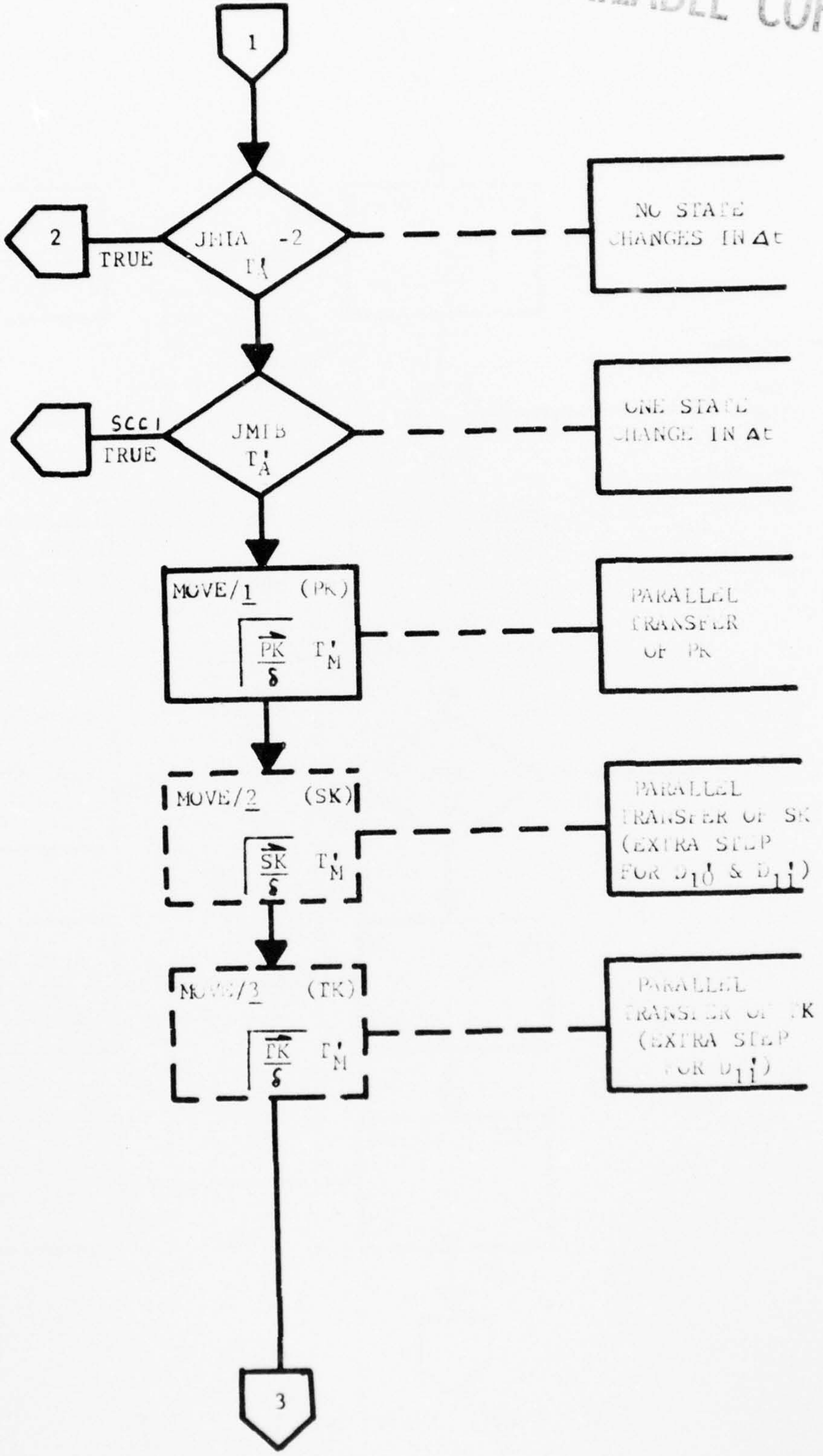


Figure B-13b. Algorithm D_9', D_{10}', D_{11}'

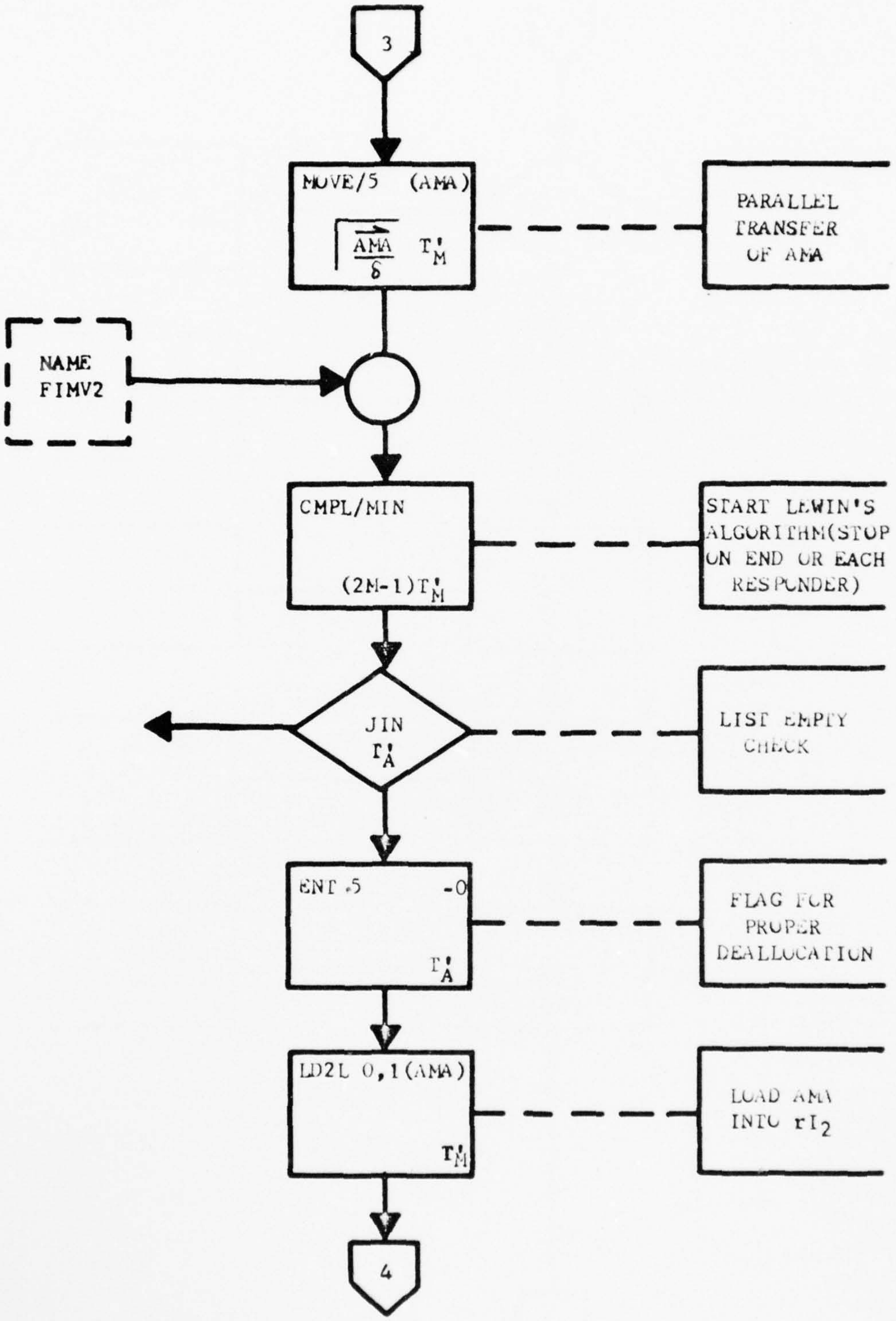


Figure B-18C. Algorithm D'_9, D'_{10}, D'_{11}

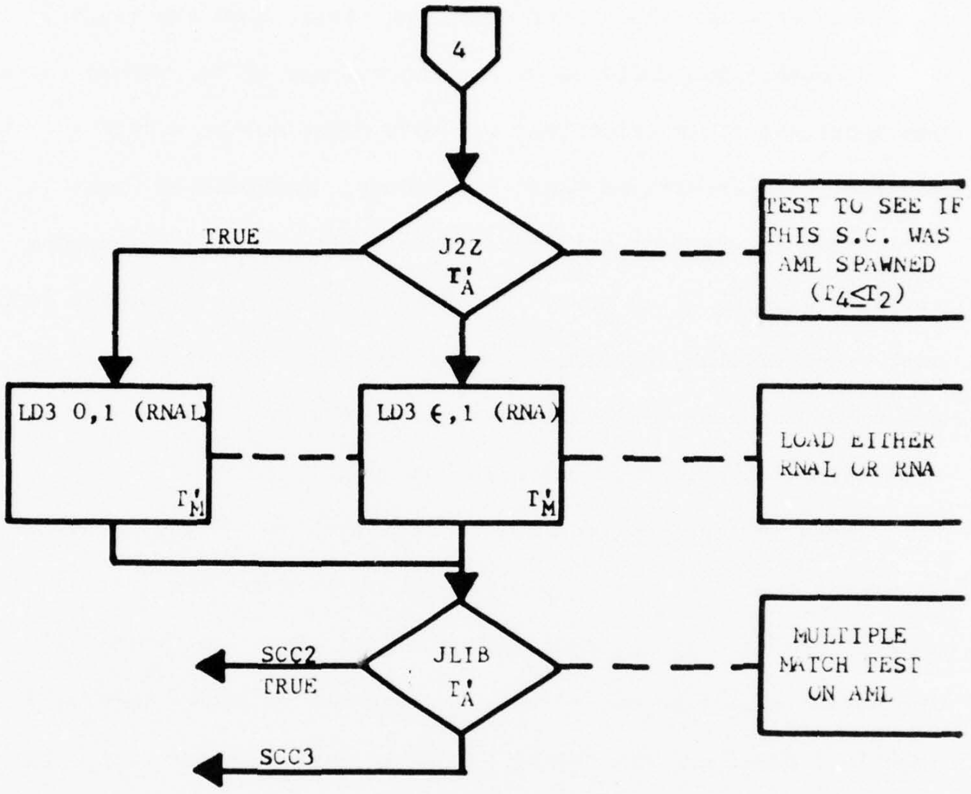


Figure B-18D. Algorithm D'_9, D'_{10}, D'_{11}

a statement about retrieving M nodes in order, one can not make the same statement about retrieving the first, then the second, and so forth. One could substitute an average of two memory cycles per retrieval (identification) but this might not be accurate under certain conditions that could occur. And finally there is additional programming associated with various follow-on aspects to the selection of an entry in the AML via Lewin's algorithm that must be considered and timed. For these reasons, the concept of the composite node cycle has been modified slightly to consider three situations in terms of this research.

These three situations can be described as follows. The first situation is the normal composite node cycle where the initial PK search returns only one node (cases 1a, 2a and 3a in Table B-4). The second situation occurs when the initial PK search returns M identical nodes and the timing listed in Table B-4 for cases 1b, 2b and 3b reflects the amount of time to select the nodes. The third situation occurs when the initial PK search returns M dissimilar nodes (cases 1c, 2c and 3c in Table B-4). These and other situations will be discussed after an explanation of the algorithms.

These algorithms differ from previous algorithms in several ways beyond those mentioned above. One, they have named entry points shown in dotted squares. Secondly, they are longer; and third, they do involve data transfer since the data is transferred between memories. They start with the same preamble as before, the subroutine linkage followed by the list selection and list empty steps. The next step is to clear the flag stored in index register five. This flag is used to select the proper deallocate sequence in DA2. At this point the actual algorithm starts. It is assumed that the actual simulation time, PK, is maintained in register X. The next step then increments register X by some amount, delta t. Now consider before proceeding that there are four times involved with understanding this algorithm. The first, ST_1, is the current simulation time. ST_2 is the sum of ST_1 and delta t, an arbitrary time increment. ST_3 is the state change time of the earliest state change lying between ST_1 and ST_2. And ST_4 is the state change time of any new state changes spawned by the state change occurring at ST_3.

At this point in the algorithm register X contains ST_2. A search on less than or equal is made on ST_2 which returns as responders all state changes occurring in delta t. If no responders are present this fact is detected by the JMIA jump instruction which cycles the program through another delta t. If there is only one responder, this is detected by the next jump instruction and this completes the instruction sequence for subcase a, described above. Subcase a terminates with an SCC1 exit, which indicates to the TFM control mechanism that there is only one responder. The timing in Table B-4 is therefore the same for subcase 1a, 2a and 3a, since the decision on a single responder is made by the fixed increment portion of the FIMV TFM, which is concerned only with LN and PK. If there are multiple responders a parallel transfer occurs of the primary key field to the AML using the modified MOVE instruction.

The length of time for this parallel transfer is a function of δ, the transfer width parameter. Parallel transfer is used in the sense that all bits of the same field in all selected nodes (selected in the search) are transferred at one time. At this point the differences among the three algorithms appear. Additional MOVE instructions are inserted for the SK and the TK respectively if they are present. The algorithm could be rearranged so that these additional transfers only occur if the PK can not be used for resolution; but without some empirical experience with an actual problem this approach seemed best. At this time it can be seen why the AML is configured differently from the AM or RAM in the sense that it has a few very wide words. The width is required so that all key bits can be contiguous to preserve the fact that the retrieval time is independent of the width. Further, additional information must be contained although not searched in the form of the RNAL and the AMA, the latter being the associative memory address, so that a reference can be made back to the buddy block of the AM node being transferred. Only a few words are needed because one does not expect on the average that a very dense state change situation will exist and if it did, delta t could be reduced as appropriate. As an aside, it should also be pointed out that in a queuing situation with a batch service discipline where one would expect several state changes to be serviced simultaneously, it was necessary to treat them separately. Each node could be serial numbered uniquely and this serial number can be added to any other keys to create a non-duplicate situation.

After the completion of all the field moves, Lewin's algorithm is started just past the second entry point (FIMV 2). The algorithm proceeds until the first stop occurs at the completion of the first partition (see Wolinsky [73] or Feng [19] for a detailed explanation). At this point a list empty check is made mainly for later cycles using the second entry point. For subcase b where multiple identical responses occur, Lewin's algorithm would detect this fact in one compare cycle, not 2M-1. The next instruction sets the deallocate flag to minus zero to indicate that there is at least one entry left in the AML. The next step loads the AMA field into index register two, but it is not known if this particular selected node was spawned by a previous AML state change such that its state change time (previous ST_4) fell below ST_2. If it was such a node then the AMA field would be zero, either from initial clearing of the memory or by a subsequent deallocation algorithm (DA_2^1). Then either RNA or RNAL would be loaded. The last step before branching is to notify the control algorithm whether there are multiple responses within the AML. The last step shown is the normal return for situation one. In practice the first use of D_9', D_{10}' or D_{11}' under subcase c obtains the first minimum, that is, the next state change. This state change is then processed through the remainder of the composite node cycle. The TFM then returns to D_9', D_{10}' or D_{11}' via the FIMV2 entry point to select the next minimum. The composite node cycles proceed until all M nodes are recovered. Therefore Lewin's algorithm is timed at 2M-1 compares and each following instruction is counted M times for Table B-4.

Situation two, M idential nodes, mentioned above, terminates with a jump to SCC2 while situation three, M dissimilar nodes, jumps to SCC3. Other situations can occur and although they are not addressed in the formal research because of complexity, they are mentioned here for completeness. (Follow-on work is needed here.) One situation can occur where a node can be inserted within the AML before it is emptied, a situation discussed previously. In this case Lewin's algorithm could either be restarted or perhaps simply continued since it can be guaranteed that the new entry will have a key of greater than or equal to any existing key. The second situation comes up when there are some duplicates among the AML entries. Based on Lewin's algorithm and Wolinsky's proof [73] the algorithm could proceed in the same manner but now instead of partitioning individual entries it will partition groups of equals along with groups of singles. Duplicates are tested for anyway by the match indicator jump. Additional comments regarding the resolution of multiple responses with Lewin's algorithm can be found in Feng [19].

Algorithms DA_1' and DA_2'

Algorithm DA_1' is simply STZ 0,1 (bb) which clears the busy bit in the AM. Algorithm DA_2' is considerably more complicated and is designed to work in conjunction with the AML algorithms. The algorithm is shown in Figure B-19 and starts out by setting up the subroutine linkage. This is followed by a flag test to determine whether situation one is in effect, which only requires a simple deallocation of the AM. Otherwise the AML node is deallocated and

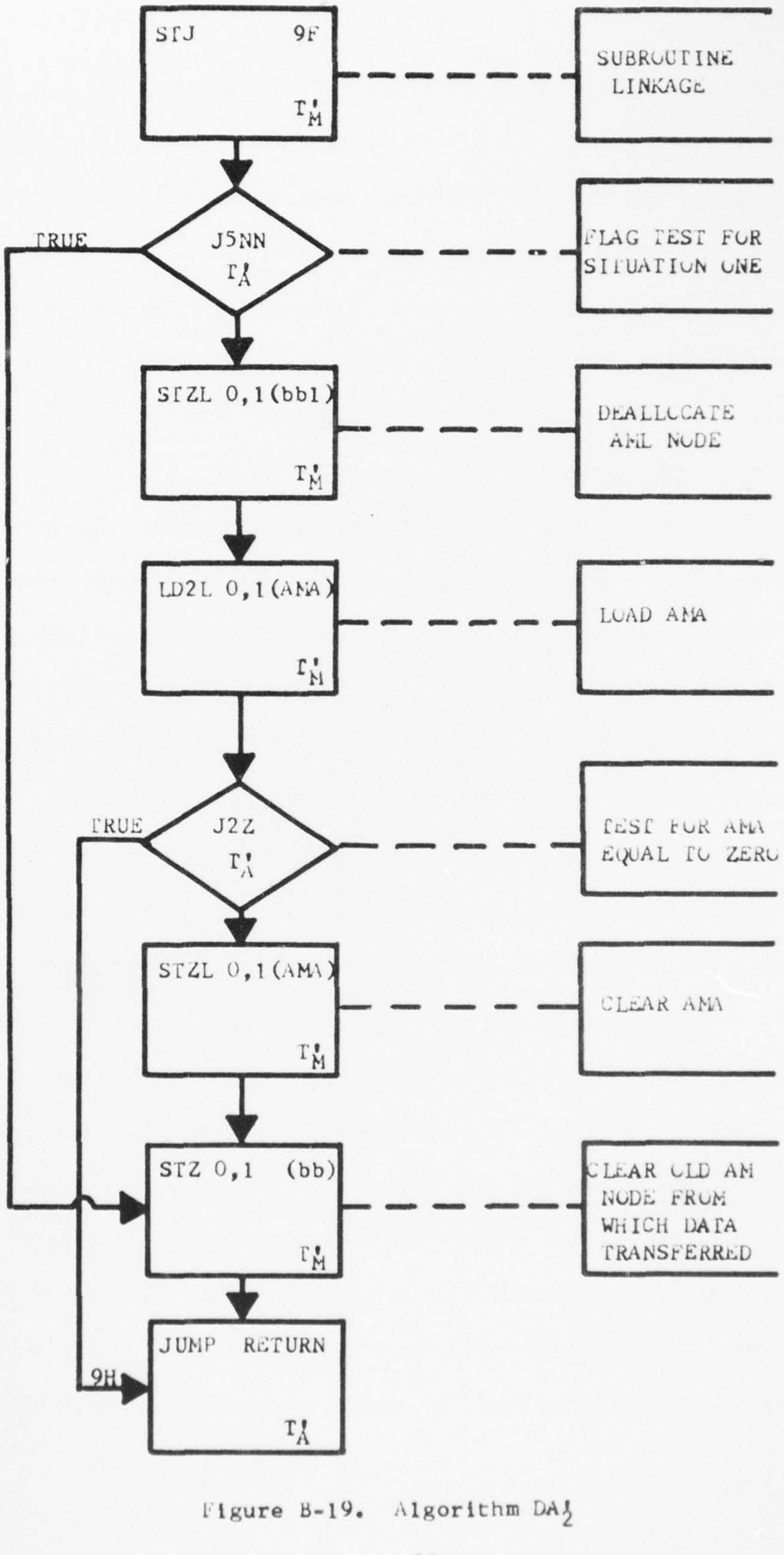


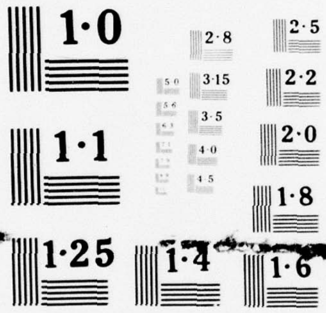
Figure B-19. Algorithm DA_2'

AD-A044 213 ROME AIR DEVELOPMENT CENTER GRIFFISS AFB N Y F/G 9/2
A COMPARISON OF SEQUENTIAL AND ASSOCIATIVE COMPUTING OF PRIORIT--ETC(U)
AUG 77 B M LANDSON
UNCLASSIFIED RADC-TR-77-221 NL

3 OF 3
ADA
044213

END
DATE
FILMED
10-77
DDC

END
DATE
FILMED
10-77
DDC


1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

AMA is loaded into index register two from the AML. A test is made to determine whether this address is zero, indicating the node was spawned into the AML directly without being transferred from the AM. If it was not transferred the deallocation process is complete. If it was transferred, then the old node in the AM must be deallocated along with clearing the AMA field. The algorithm terminates with a jump return.

Search Algorithms S'_1, S'_2, S'_3, and S'_4

The search algorithms are set up in the same manner as their random access counterparts. The three options of find first (comparand), find all (comparand) and find first minimum or maximum are shown in Figure B-20. Although there are three options there are four algorithms, S'_1, S'_2, S'_3 and S'_4. This occurs as a result of the nature of the associative architecture, as follows. Search algorithms S'_1 and S'_3 are used for find first and find all for the AM and AM/RAM architectures respectively. Since the associative memory searches all words in parallel and has match indicators which record all ties or equals, the same algorithm satisfies both the find first and the find all search criteria. In the AM/RAM architecture, however, there is an extra step (LD2) which results in a slightly different timing for S'_3. S'_2 and S'_4 are the algorithms for find the minimum or maximum for the AM and AM/RAM respectively. The difference for minimum or maximum lies in which is used in the CMP// instruction. Minimum is shown in the figure. Again S'_4 differs from S'_2 by the addition of the LD2 command which alters the timing slightly. In all cases there is a preface which selects the list and performs a list empty check. This preface is shown at the top of Figure B-20. The algorithm timings are shown in Figure B-4.

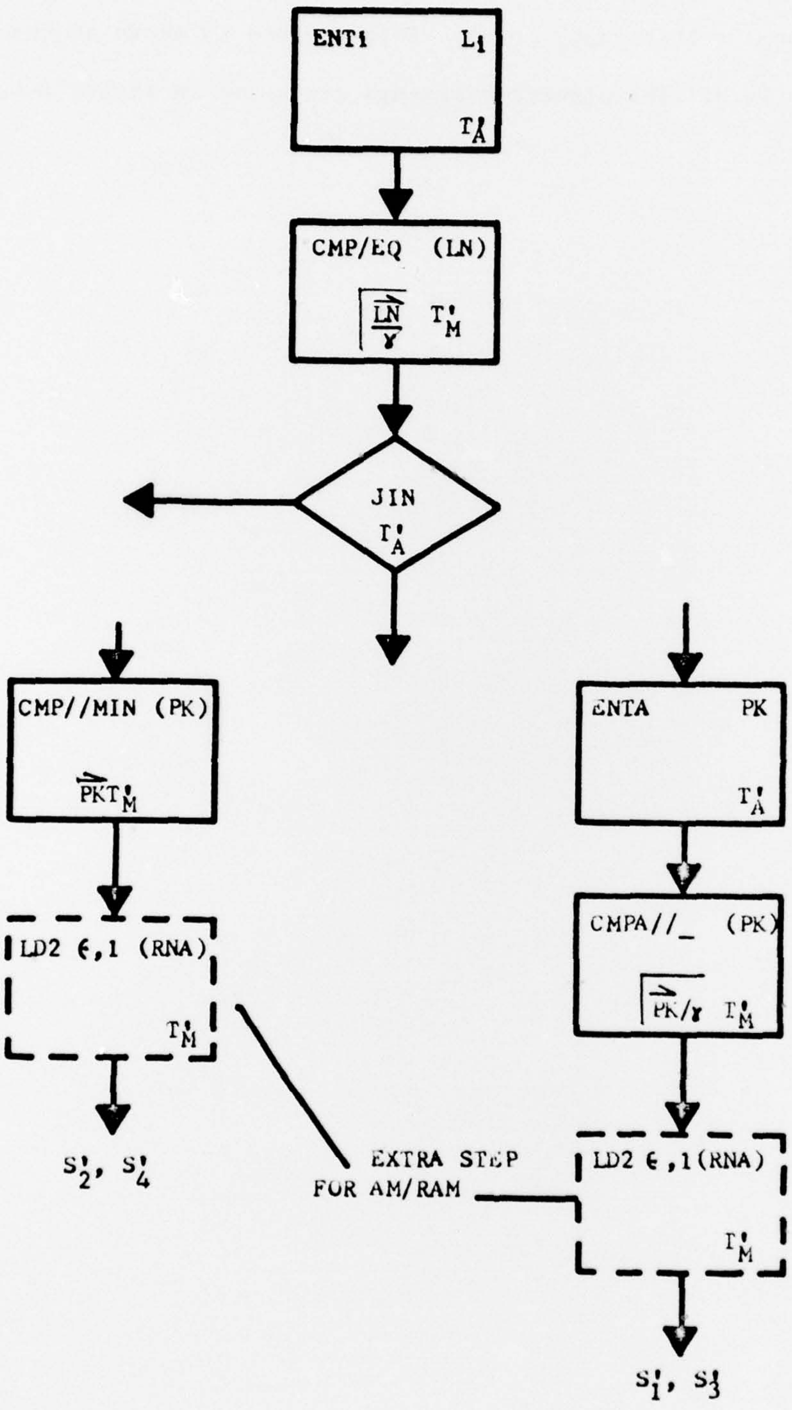


Figure B-20. Algorithm S'_1, S'_2, S'_3, S'_4

BIBLIOGRAPHY

1. A. T. Berztiss, Data Structures: Theory and Practice, New York: Academic Press, 1971.

2. W. J. Bouknight and others, "The ILLIAC IV System," Proceedings of the IEEE, Vol. 60, No. 4:369-388 (April 1972).

3. G. E. P. Box and G. M. Jenkins, Time Series Analysis Forecasting and Control, San Francisco: Holden-Day, 1971.

4. D. Brotherton and S. Domchick, Preliminary Programming Manual for RADC 2048 Word Associative Memory, Akron, Ohio: Goodyear Aerospace Corporation, 1966.

5. J. N. Buxton, Simulation Programming Languages, Amsterdam, Netherlands: North Holland Publishing Company, 1968.

6. E. Chapin, "A Comparison of File Organization Techniques," ACM 24th National Conference, 1969: pp. 273-283.

7. -----, Computers, A Systems Approach, New York: Van Nostrand Reinhold Company, 1971.

8. R. W. Conway and others, "Some Problems of Digital Systems Simulation," Management Science, pp. 92-110 (October 1959).

9. R. B. Cooper, Introduction to Queuing Theory, New York: The Macmillan Company, 1972.

10. D. R. Cox and W. C. Smith, Queues, London: Methuen and Co., Ltd., 1961.

11. E. W. Davis, A Multiprocessor for Simulation Applications, Urbana, Illinois: University of Illinois at Urbana, Dept. of Computer Science Ph. D. Thesis, 1972.

12. C. R. DeFiore, An Associative Approach to Data Management, Syracuse, New York: Syracuse University, Dept. of Systems and Information Science Ph. D. Thesis, 1972.

13. M. D'Imperio, "Data Structures and Their Representation in Storage," Annual Review In Automatic Programming, Vol. 5:1-75.

14. G. Dodd, "Elements of Data Management Systems," Computing Surveys - ACM, pp. 117-133 (June 1969).

15. N. R. Draper and H. Smith, Applied Regression Analysis, New York: John Wiley & Sons, Inc., 1966.

16. J. Emshoff and R. Sisson, Design and Use of Computer Simulation Models, New York: The Macmillan Company, 1970.

17. Feng, Tse-yun and others, Associative Processor Computing System, Part II, Associative Processor Application, Syracuse, New York: Syracuse University, 1971.

18. Feng, Tse-yun, Associative Processor Computing System, Part III, Bibliography on Associative Processors, Syracuse, New York: Syracuse University, 1971.

19. -----, Large Scale Information Processing Systems, Vol. V, Study of Associative Memory Systems, RADC TR 70-80, AD#708725-29 Griffiss A.F.B., New York: Rome Air Development Center, 1970.

20. -----, Parallel Processor Characteristics and Implementation of Data Manipulating Functions, Department of Electrical and Computer Engineering Technical Report TR 73-1, Syracuse, New York: Syracuse University, 1973.

21. -----, "Search Algorithms for Associative Memories," Princeton University Conference on Information Science and Systems Proceedings, pp. 422-426 (March 1970).

22. G. Fishman, Concepts and Methods in Discrete Simulation, New York: John Wiley & Sons, Inc., 1973.

23. M. J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Electronic Computers, Vol. C-21, No. 9:948-960 (September 1972).

24. S. H. Fuller, Orthogonal Versus Array Computing, Digital Systems Laboratory CCS-961, Tech Note No. 4, Stanford, California: Stanford Electronics Laboratory, Stanford University.

25. A. Gafarian and C. Ancker, "Mean Value Estimation from Digital Computer Simulation," Operations Research, Vol. 14, No. 1:25-44 (January-February 1966).

26. Geoffrey Gordon, System Simulation, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1969.

27. G. R. Grossman and D. C. Scafe, SSK/SSK Reference Manual, ILLIAC IV Document No. 178, Urbana, Illinois: University of Illinois at Urbana, June 1969.

28. A. G. Hanlon, "Content-Addressable and Associative Memory Systems--A Survey," IEEE Transactions on Electronic Computers, Vol. EC-15, No. 4:509-521 (August 1966).

29. H. Hellerman, Digital Computer System Principles, New York: McGraw-Hill Book Company, 1967.

30. D. A. Hodges, Editor, Semiconductor Memories, New York: IEEE Press, 1972.

31. C. R. Hyde, An Overview of Associative Memories, ILLIAC IV Document No. 237, Urbana, Illinois: University of Illinois at Urbana, 1971.

32. N. K. Jaiswal, Priority Queues, New York: Academic Press, 1968.

33. P. W. M. John, Statistical Design and Analysis of Experiments, New York: The Macmillan Company, 1971.

34. H. Katzan, Advanced Programming, New York: Van Nostrand Reinhold Company, 1970.

35. P. J. Kiviat, "Simulation Languages," in Computer Simulation Experiments with Models of Economic Systems, edited by T.H. Naylor, New York: John Wiley & Sons, Inc., 1971, pp. 406-486.

36. D. Knuth, The Art of Computer Programming, Vol. I: Fundamental Algorithms, Reading, Massachusetts: Addison Wesley Publishing Company, 1968.

37. -----, MIX, Reading, Massachusetts: Addison Wesley Publishing Company, 1969.

38. -----, The Art of Computer Programming, Vol. II: Seminumerical Algorithms, Reading, Massachusetts: Addison Wesley Publishing Company, 1969.

39. -----, The Art of Computer Programming, Vol. III: Sorting and Searching, Reading, Massachusetts: Addison Wesley Publishing Company, 1972.

40. L. J. Koczela, "The Distributed Processor Organization," Advances in Computers, Volume 9, edited by Franz L. Alt and Morris Rubinoff, New York: Academic Press, 1968.

41. D. Kroft, An Associative Memory Computer with an Application to List Processing, Ph. D. Thesis, New York: Columbia University, 1969.

42. D. J. Kuck and others, "Measurements of Parallelism in Ordinary Fortran Programs," Proceedings of the 1973 Sagamore Computer Conference, Information Sciences Division of the Rome Air Development Center, Griffiss A.F.B., New York, 1973.

43. -----, "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speed-up," IEEE Transactions on Computers, Vol. C-21, No. 12 (December 1972).

44. Roy E. Lave, Jr., "Time Keeping for Simulation," in Journal of Industrial Engineering (July 1967).

45. M. H. Lewin, "Retrieval of Ordered Lists from a Content Addressed Memory," RCA Review, Vol. 23:215-229 (June 1962).

46. G. J. Lipovski, "The Architecture of a Large Associative Processor," 1970 Spring Joint Computer Conference, in AFIPS Proceedings, 39:385-396, Washington, D.C.: Thompson Book Company, 1970

47. -----, "On Data Structures in Associative Memories," in Proceedings of the Symposium on Data Structure in Programming Languages, pp. 346-365, ACM-Sigplan Publication, February 1971.

48. N. C. Machado, An Array Processor with a Large Number of Elements, Center for Advanced Computation Document No. 25 UIU CDCS/R-72-499, Urbana, Illinois: University of Illinois at Urbana, January 1972.

49. J. Minker, "Bibliography #25: An Overview of Associative or Content Addressable Memory Systems and a KWIC Index to the Literature 1958-1970," University of Maryland and Auerbach Corporation Computing Reviews, pp. 453-524 (October 1971).

50. H. Morgan and G. Siegel, Synchronization Models for Simulation and List Processing, Technical Report No. 113, Department of Operations Research, Ithaca, New York: Cornell University, June 1970.

51. J. C. Murtha, "Highly Parallel Information Processing Systems," in Advances in Computers, Volume 7, edited by Franz L. Alt and Morris Rubinoff, New York: Academic Press, 1966, pp. 2-116.

52. R. E. Nance, "On Time Flow Mechanisms for Discrete System Simulation," Management Science, Vol. 18, No. 1 (September 1971).

53. T. H. Naylor and others, Computer Simulation Techniques, New York: John Wiley & Sons, Inc., 1966.

54. T. H. Naylor, Editor, The Design of Computer Simulation Experiments, Durham, North Carolina: Duke Press, 1969.

55. T. H. Naylor, Computer Simulation Experiments with Models of Economic Systems, New York: John Wiley & Sons, Inc., 1971.

56. V. Orlando, Associative Processing in the Solution of Network Problems, Ph. D. Dissertation, Syracuse, New York: Syracuse University, 1972.

57. B. Parhami, "Associative Memories and Processors: An Overview and Selected Bibliography," Proceedings of the IEEE, Vol. 61, No. 6:722-730 (June 1973).

58. E. Parzen, Stochastic Processes, San Francisco: Holden-Day, 1962.

59. C. Peters, Associative Memory Compiler Techniques Study, RADC TR 67-474, Rome, New York: Rome Air Development Center, 1967.

60. J. Posdamer and others, "The Application of Associative Processing in Discrete Simulation," Fifth Annual Conference on the Application of Simulation, New York, December 1971.

61. A. A. B. Pritsker, The GASP IV Simulation Language, New York: John Wiley & Sons, Inc., 1974.

62. A. A. B. Pritsker and P. J. Kiviat, Simulation with GASP II, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1969.

63. J. A. Rudolf, "A Production Implementation of an Associative Array Processor - Staran," AFIPS Proceedings, Vol. 41, Part 1, pp. 229-241 (1972), Washington, D.C.: Thompson Book Company.

64. W. Shooman, "Parallel Computing with Vertical Data," 1960 Fall Joint Computer Conference, in AFIPS Proceedings, 18: 111-116, Washington, D.C.: Thompson Book Company, 1960.

65. -----, "Orthogonal Processing," Proceedings of the Symposium on Parallel Processing Systems, Technologies, and Applications, June 26-28, 1969.

66. H. Stone, "Associative Processing for General Purpose Computers Through the Use of Modified Memories," 1968 Fall Joint Computer Conference in AFIPS Proceedings, 33:949-955, Washington, D.C.: Thompson Book Company, 1968.

67. D. Teichroew and J. F. Lubin, "Computer Simulation--Discussion of the Technique and Comparison of Languages," Communications of the ACM, Vol. 9, No. 10:723-741 (October 1966).

68. J. K. Thurber and R. O. Berg, "Applications of Associative Processors," Computer Design, pp. 103-110 (November 1971).

69. J. G. Vaucher and P. Duval, "A Comparison of Simulation Event List Algorithms," Communications of the ACM, Vol. 18, No. 4: 223-230 (April 1975).

70. A. Weinberger, "The Hybrid Associative Memory Concept," Computer Design, pp. 77-85 (January 1971).

71. W. K. Wickham, Jr., Time Flow Mechanisms for Discrete Simulation Models, Ph. D. Thesis, Southern Methodist University, 1971.

72. B. Wittman, "An Investigation into Information Storage and Retrieval Utilizing ILLIAC IV," Center for Advanced Computation Document No. 22, Urbana, Illinois: University of Illinois at Urbana, November 1971.

73. A. Wolinsky, "A Simple Proof of Lewin's Ordered Retrieval Algorithm for Associative Memories," Communications of the ACM, Vol. 11, No. 7:488-490 (July 1968).

74. -----, "Principles and Applications of Associative Memories," Third Annual Symposium on the Interface of Computer Science and Statistics, Los Angeles, California, January 1969.

7